

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI[®]

System Support for Pervasive Applications

Robert Grimm

A dissertation submitted in partial fulfillment
of the requirements for the degree of

Doctor of Philosophy

University of Washington

2002

Program Authorized to Offer Degree: Computer Science and Engineering

UMI Number: 3072090

Copyright 2002 by
Grimm, Robert Otto

All rights reserved.

UMI^a

UMI Microform 3072090

Copyright 2003 by ProQuest Information and Learning Company.

All rights reserved. This microform edition is protected against
unauthorized copying under Title 17, United States Code.

ProQuest Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346

© Copyright 2002

Robert Grimm

In presenting this dissertation in partial fulfillment of the requirements for the Doctoral degree at the University of Washington, I agree that the Library shall make its copies freely available for inspection. I further agree that extensive copying of this dissertation is allowable only for scholarly purposes, consistent with "fair use" as prescribed in the U.S. Copyright Law. Requests for copying or reproduction of this dissertation may be referred to Bell and Howell Information and Learning, 300 North Zeeb Road, Ann Arbor, MI 48106-1346, to whom the author has granted "the right to reproduce and sell (a) copies of the manuscript in microform and/or (b) printed copies of the manuscript made from microform."

Signature

A handwritten signature in black ink, consisting of a series of loops and a long horizontal stroke at the end.

Date

11/1/02

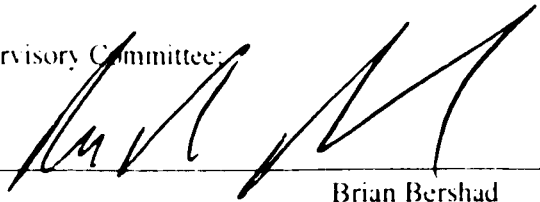
University of Washington
Graduate School

This is to certify that I have examined this copy of a doctoral dissertation by

Robert Grimm

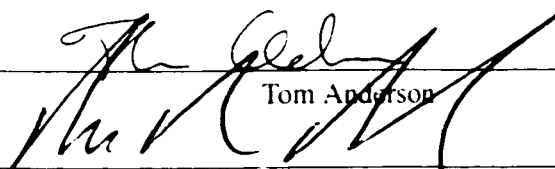
and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

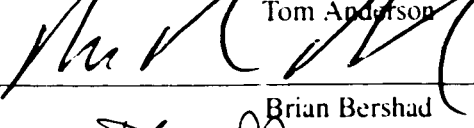
Chair of Supervisory Committee:

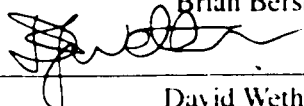


Brian Bershad

Reading Committee:



Tom Anderson


Brian Bershad


David Wetherall

Date:

12/19/02

University of Washington

Abstract

System Support for Pervasive Applications

by Robert Grimm

Chair of Supervisory Committee:

Professor Brian Bershad
Computer Science and Engineering

Pervasive computing provides an attractive vision for the future of computing. Computational power will be available everywhere. Mobile and stationary devices will dynamically connect and coordinate to seamlessly help people in accomplishing their tasks. For this vision to become a reality, developers must build applications that constantly adapt to a highly dynamic computing environment. However, existing distributed systems technologies are ill-suited for building adaptable applications. To make the developers' task feasible, we introduce a system architecture for pervasive computing, called *one.world*. Our architecture provides an integrated and comprehensive framework for building pervasive applications. It includes services, such as service discovery and migration, that help to build applications and directly simplify the task of coping with constant change. We describe the design and implementation of our architecture and reflect on our own and others' experiences with using it.

TABLE OF CONTENTS

List of Figures	iv
List of Tables	vi
Chapter 1: Introduction	1
1.1 This Dissertation	3
1.1.1 Research Contributions	5
1.1.2 Limitations	6
1.1.3 Dissertation Organization	7
Chapter 2: Motivation and Approach	8
2.1 The Unique Requirements of Pervasive Computing	10
2.2 Adaptability and Transparency	13
2.3 The Biology Laboratory as an Example Application Domain	15
2.3.1 Short-comings of the Status Quo	18
Chapter 3: Architecture	20
3.1 Foundation Services	20
3.2 System Services	22
3.3 Library Support	23
3.4 The Big Picture	24
3.5 <i>one.world</i> and Distributed Systems Concerns	25
Chapter 4: Programming Model	27
4.1 Namespaces	28
4.2 Data Management	29

4.2.1	Query Language and Query Engine	30
4.2.2	Structured I/O	32
4.3	Execution Model	34
4.3.1	Interacting with the Kernel and Other Environments	37
4.3.2	Reliably Managing Event Exchanges	39
4.4	Communication Model	41
4.5	Application Persistence	46
Chapter 5:	Implementation	51
5.1	Environments as Application Containers	53
5.2	Reliable Event Exchanges through Operations	57
5.3	Discovery and Remote Event Passing	61
5.4	Checkpointing and Migration	65
5.5	Summary	70
Chapter 6:	Experimental Evaluation	71
6.1	Completeness	72
6.1.1	Replication Service	72
6.1.2	Emcee and Chat	74
6.1.3	Discussion	77
6.2	Complexity	77
6.3	Performance	80
6.4	Utility	84
6.4.1	Building Distributed and Pervasive Applications	84
6.4.2	Labscape	86
Chapter 7:	Discussion and Future Work	91
7.1	Data Model	93
7.2	Event Processing	94

7.3	Leases	96
7.4	A Unified Interface to Storage and Communications	96
7.5	User Interface	97
7.6	Interacting with the Outside World	98
7.7	Outlook	99
Chapter 8:	Related Work	102
Chapter 9:	Conclusions	109
	Bibliography	112
Appendix A:	The Migration Protocol	126

LIST OF FIGURES

1.1	A sampling of unconventional computing devices	2
2.1	Illustration of our approach	12
2.2	A workbench in a biology laboratory	15
2.3	A screenshot of Labscape’s user interface	16
3.1	Overview of <i>one.world</i> ’s architecture	21
3.2	Illustration of an example environment hierarchy	22
3.3	The big picture	24
4.1	Definition of a tuple	31
4.2	The event handler interface	34
4.3	Illustration of the relationship between imported and exported event handlers	35
4.4	Code example for initializing an application	36
4.5	Illustration of the request/monitor mechanism	38
4.6	Illustration of the operation library	40
4.7	Code example for creating an operation	41
4.8	Code example for sending a remote event	45
4.9	Code examples for checkpointing, restoring, and moving an environment	50
5.1	Illustration of Emcee’s fetcher protocol	59
6.1	Illustration of our replication service’s structure	73
6.2	A screenshot of Emcee’s user interface	74
6.3	A screenshot of Chat’s user interface	76
6.4	Discovery server throughput under an increasing number of receivers	81

6.5	Discovery server throughput under an increasing number of senders	82
6.6	Audio messages received by Chat in a changing runtime environment	83
6.7	Illustration of Labscape's structure	88
A.1	Illustration of <i>one.world</i> 's migration protocol	127

LIST OF TABLES

3.1	Application needs and corresponding system services	22
3.2	<i>one.world</i> and distributed systems concerns	26
4.1	The environment operations	29
4.2	The structured I/O operations	32
4.3	Options for exporting event handlers to remote event passing and discovery	44
5.1	Summary of implementation challenges	69
6.1	Breakdown of development times in hours for Emcee and Chat	79
8.1	Comparison of migration services	106

ACKNOWLEDGMENTS

First and foremost, I thank my advisor Brian Bershad for his guidance over the last several years. I benefitted tremendously from his insight, his uncanny ability to always ask the right questions, and his passion for conducting research that matters. I greatly enjoyed working with you, Brian. Thank you!

I also thank Tom Anderson, Gaetano Borriello, Steven Gribble, and David Wetherall (who graciously agreed to become my co-advisor during Brian's leave from the University of Washington) for their advice, insight, and support. I very much appreciate their help with conducting and presenting my research, advising other students, and creating and teaching a class.

one.world (obviously) represents the result of a larger group effort and I thank all other group members for their contributions. In particular, I greatly enjoyed collaborating with Janet Davis and Eric Lemar on this project. Both helped refine my original design (in often lively discussions) and contributed significantly to *one.world*'s implementation, our applications, and the evaluation. Janet focused on networking, remote events, replication, and streaming audio, while Eric focused on tuple storage, discovery, and measurements. Adam MacBeth and Steven Swanson also made important contributions to the architecture, with Adam concentrating on discovery and Steven concentrating on replication and how to port *one.world* to smaller devices, such as the iPaq handheld computer. In addition, Ben Hendrickson implemented parts of tuple storage, Kaustubh Deshmukh implemented a debugger, and Daniel Cheah implemented a web server running on top of *one.world*. I also thank the students of University of Washington's CSE 490dp for serving as test subjects and Vibha Sazawal and David Notkin for their advice and assistance in evaluating the students' projects.

I thank Mike Swift for the many spirited discussions on the design of *one.world* and for his insightful and detailed feedback on papers describing the work. Moreover, our work on structured storage [45] provided a first step towards tuple-based storage in *one.world*.

The University of Washington's Department of Computer Science and Engineering provided an

ideal environment for conducting my research. Faculty, students, and staff alike have created an incredibly friendly and collaborative work environment. I will miss this place!

My work was funded in part under DARPA contracts F30602-98-1-0205 and N66001-99-2-892401, by IBM Corporation and Intel Foundation graduate fellowships, and by an Intel Corporation internship.

Last, but certainly not least, I owe immense gratitude to my family—notably my parents, Rosa and Otto Grimm—and my friends—especially Bernard, James McNamara, Michael Protevi, and Lance Trotter—for their loving support over the last several years.

DEDICATION

“Quidquid latet, apparebit:
Nil inultum remanebit.”

Chapter 1

INTRODUCTION

Pervasive, or ubiquitous, computing [113] has the potential to radically transform the way people interact with computers. The key idea behind pervasive computing is to deploy a wide variety of computing devices throughout our living and working spaces. These devices coordinate with each other and network services [27], with the goal of providing people with universal access to their information and seamlessly assisting them in completing their tasks. Pervasive computing thus marks a major shift in focus, away from the actual computing technology and towards people and their needs. So, instead of manually managing their computing environment by, for example, copying files between devices or converting between data formats, users “simply” access their applications and their data whenever and wherever they need.

With its vision of ubiquitous information access, pervasive computing significantly impacts computing devices and their deployment. In addition to conventional desktop and server computers, pervasive computing environments encompass many different devices of various sizes and capabilities, including PDA cell phones, video game consoles, and robotic dogs (see Figure 1.1). We can reasonably expect the variety of devices to further increase over time. Furthermore, in addition to well-administered and -controlled computing laboratories and server rooms with racks of computers, computing devices are now everywhere, often embedded in places not typically associated with computing, such as living rooms or biology laboratories.

Pervasive computing also changes how people interact with the surrounding computing infrastructure. First, in contrast to conventional computing environments, people focus on their activities and not on the computers. Furthermore, as already pointed out, these computers are often embedded within the landscape. For example, in a biology laboratory, researchers focus on their experiments and not on the computing devices used to capture experimental results, such as digital pipettes or



Figure 1.1: A sampling of unconventional computing devices. Pervasive computing environments encompass a wide variety of computing devices, such as (clockwise from the upper left corner) MP3 players, PDA cell phones, tablet computers, computerized wrist watches, robotic dogs, and video game consoles. All pictured devices are either commercial prototypes or already shipping; we can reasonably expect the variety of devices to further increase over time.

incubators. Second, tasks often last days and may span many devices, people, and places. Moreover, task requirements may change frequently. For example, biology experiments often take hours, if not days, to complete and involve many collaborating researchers working at different laboratory stations. As an experiment progresses, researchers may schedule additional steps, for instance, to determine whether an unexpected outcome was caused by contaminants.

The key challenge for developers is to build pervasive applications that continuously adapt to such a highly dynamic environment and continue to function even if people move through the physical world or switch devices, and if the surrounding network infrastructure provides only limited services. However, existing approaches to building distributed applications, including client/server or multitier computing, are ill-suited to meet this challenge.

The fundamental problem is that these approaches try to hide distribution and rely on technologies, such as remote procedure call (RPC) packages [11] or distributed file systems [66], that extend single-node programming methodologies to distributed systems. Because these technologies hide

remote interactions, favor static composition through programmatic interfaces, and often encapsulate data and functionality in the form of objects, they make it hard to anticipate failures, to extend applications, and to share and search data. Consistent with the push towards hiding distribution, applications built on top of these technologies tend to be structured like single-node applications and assume an execution environment where resources are constant and continuously available.

As a result, users are forced to “stitch up the seams” and need to explicitly reconfigure their computers every time the execution environment changes. For example, with today’s wireless networking technologies, people need to manually adapt their computers every time they enter a different network. Existing systems and applications have no notion of “entering a new network” and thus need to be explicitly configured with the wireless network name and access key, to say nothing of necessary file servers or close-by printers. However, forcing users to adapt is impractical and, fundamentally, antithetical to the vision of pervasive computing.

1.1 This Dissertation

This dissertation explores how to build applications for pervasive computing environments. It introduces a system architecture, called *one.world*, that provides an integrated framework that has been specifically designed for building adaptable applications, with the goal that applications adapt to change instead of users.

The hypothesis behind this work is that, by focusing on the unique requirements of pervasive computing, effective system support for pervasive applications becomes possible. The idea is to design and implement a system architecture that focuses on meeting the requirements ill-served by contemporary systems. By focusing on these requirements, our system architecture—in contrast to many previous systems—lets applications instead of users adapt to change, yet—like previous systems—does not place an undue burden on developers. To test this hypothesis, we validate the architecture with application builders and gain actual experience with real pervasive applications. Based on that experience, we iterate the design process and address the additional, empirically determined needs of pervasive applications and their developers. In other words, we rely on a design methodology that starts out by focusing on the unique requirements and then iteratively refines the design based on empirical evidence, resulting in a practical system architecture for pervasive

computing.

Our work is motivated by the insight that, in direct opposition to conventional distributed systems, system support for pervasive applications must expose distribution rather than hide it. That way, applications can see change and then adapt to it instead of forcing users to constantly reconfigure their systems. More specifically, system support for pervasive applications must meet three requirements. First, as people move throughout the physical world—either carrying their own portable devices or switching between devices—an application’s location and execution context change all the time. As a result, system support needs to *embrace contextual change* and not hide it from applications. Second, users expect that their devices and applications just plug together. System support thus needs to *encourage ad hoc composition* and not assume a static computing environment with a limited number of interactions. Third, as users collaborate, they need to easily share information. As a result, system support needs to *recognize sharing as the default*.

Individually, the three requirements have been recognized by previous distributed systems efforts. For instance, Bayou [88, 102] exposes different data values in the data stores on different devices, and Odyssey [81] relies on asynchronous notifications to expose contextual change to applications. Furthermore, the World Wide Web is built on only two basic operations, GET and POST, which simplifies dynamic composition and, in part, has enabled the addition of new services, such as caching [22, 103], content transformation [36], and content distribution [53]. Finally, the recent move towards expressing all data on the Internet as XML [15] attempts to facilitate pervasive sharing. Clearly, our approach to building pervasive applications draws on these efforts; but it differs in that we do not focus on any single requirement and address all three requirements *at the same time*. Furthermore, by exposing distribution, our approach differs from a large class of efforts that have explored how to build services that (largely) transparently adapt to a changing execution context. We view these efforts as complimentary to our own work and discuss them in the following chapter.

Our architecture, *one.world*, has been designed from the ground up to meet the three requirements at the same time. It is based on a simple programming model that relies on tuples (possibly nested records of name/value pairs) for all data, thus making it easy to share data, and asynchronous events for all communications, thus making it easy to notify applications of change. Like any distributed system, it has facilities for managing processes, storage, and point-to-point communications. More importantly, it provides a set of services, such as service discovery and migration, that

directly simplify the task of coping with constant change. Our architecture reuses existing operating system technologies where appropriate and innovates where necessary; the focus is to provide an integrated and comprehensive framework for building pervasive applications.

We have validated our architecture by supporting the Labscape project [6] in porting their digital biology laboratory assistant to *one.world*, by using our architecture as the basis for student projects in a senior-level capstone design course, and by building our own utilities and applications on top of *one.world*—notably, Emcee, our user and application manager, and Chat, a text and audio messaging system. Based on these experiences, we show that our architecture (1) enables others to successfully build pervasive applications, (2) is not significantly harder to program than with conventional programming styles, (3) is sufficiently complete to support additional services and utilities on top of it, and (4) has acceptable performance, with applications reacting quickly to changes in their runtime context. Our evaluation thus confirms the hypothesis behind this work and establishes *one.world* as a solid foundation for future work on system support for pervasive computing.

1.1.1 Research Contributions

This dissertation makes the following research contributions:

- We present a system architecture for pervasive computing, called *one.world*. Rather than providing middleware on top of an existing distributed system, our architecture has been designed from the ground up to directly address the unique requirements of pervasive computing—change, ad hoc composition, and pervasive sharing. As a result, we can avoid unnecessary complexity and provide an integrated and comprehensive framework for building pervasive applications.
- We introduce a new service for grouping applications and their data as well as for composing applications. This so-called *environment* service provides a nestable container for persistent storage and computations alike, thus representing a combination of file system directories and nested processes [17, 33, 107] in other operating systems.
- We present a practical migration service to simplify the construction of applications that follow people as they move through the physical world. By integrating our migration service

with our architecture’s environment service, we can strike a better balance between the complexity of transparent process migration, as provided, for example, by Sprite [28], and the limited utility of mobile agent systems, such as IBM’s Aglets [63].

- We present a classification of service discovery options and describe a discovery service that supports all options through an API with only three simple operations. The same API is also used for point-to-point communications. The result is an elegant and powerful interface that is considerably more flexible than, for example, RPC, yet also easy to use—even though our architecture does not hide remote interactions.
- We identify a software pattern for managing asynchronous interactions. The so-called *logic/operation pattern* structures applications into logic—computations that do not fail—and operations—interactions that may fail. The corresponding operation library simplifies such interactions by keeping the state associated with event exchanges and by providing automatic timeouts and retries.

In addition to these major research contributions, this dissertation also reflects a (subtle) cultural shift. In evaluating distributed systems, previous work has often focused on system performance as the primary evaluation metric. While we still consider performance an important evaluation metric, we believe that it cannot be the only one. Notably, when introducing a new system architecture, an important question is whether developers can effectively build applications on top of that architecture. As a result, we also draw on other metrics, such as programmer productivity, to evaluate our architecture, thus expanding the repertoire of metrics for systems work.

1.1.2 Limitations

Consistent with the hypothesis, the design of *one.world* focuses on meeting the unique requirements of pervasive computing. As a result, it does not address all possible needs of pervasive applications. Notably, we have not addressed how to secure pervasive applications, including how to express access control constraints and how to enforce them in a constantly changing computing environment. Clearly, security is a very important concern, though it is not a new concern. At the same time, the design *does* reflect an understanding that appropriate security mechanisms will have to be built into

the system in the future. We have thus included a simple, yet effective interposition mechanism (the request/monitor mechanism described in Chapter 4.3.1), which, as I showed in [43], can provide a solid basis for implementing access control and auditing.

Furthermore, while building *one.world* from the ground up (without employing existing middleware) has allowed us to avoid many of the limitations of existing distributed systems, it has also introduced new ones. Notably, as discussed in Chapter 7, using our own data model—based on tuples—and our own communication model—based on the exchange of events—has resulted in an architecture that has only very limited interoperability with existing Internet services. Furthermore, as discussed in Chapter 6.3, the scalability of our implementation, especially that of service discovery, is limited, making it suitable only for pervasive computing environments with several dozens of people and devices. We suggest approaches to avoiding these limitations in the respective chapters.

1.1.3 Dissertation Organization

This dissertation is structured as follows. In Chapter 2, we motivate our work and introduce our approach to building pervasive applications. Chapter 3 provides an overview of our architecture. We describe its programming model in Chapter 4 and our Java-based implementation in Chapter 5. In Chapter 6, we introduce the user-space services, utilities, and applications we and others have built and present the evaluation of our architecture, which is based on these programs. In Chapter 7, we reflect on our experiences with building and using *one.world* and suggest important areas for future work on system support for pervasive computing. Chapter 8 reviews related work. Finally, Chapter 9 concludes this dissertation.

Chapter 2

MOTIVATION AND APPROACH

From a systems viewpoint, the pervasive computing space presents the unique challenge of a large and highly dynamic distributed computing environment. This suggests that pervasive applications really are distributed applications. Yet, existing approaches to building distributed systems fall short along three main axes when considering pervasive applications. First, existing approaches make it hard to anticipate change, including failures, as they tend to hide remote interactions. Second, they make it hard to extend applications, as they favor static composition through programmatic interfaces. Third, they make it hard to share and search data, as they often encapsulate data and functionality in the form of objects. The common cause for all three shortcomings is an attempt to simplify application development by extending single-node programming methodologies to distributed systems. However, this drive also results in systems that are unsuitable for building adaptable applications, thus placing the burden of adapting to change on users. We now discuss the three shortcomings of existing approaches to building distributed systems in detail.

First, many existing distributed systems seek to hide distribution and, by building on distributed file systems [66] or remote procedure call (RPC) packages [11], mask remote resources as local resources. This transparency simplifies application development, since accessing a remote resource is just like performing a local operation. However, this transparency also comes at a cost in service quality and failure resilience. By presenting the same interface to local and remote resources, transparency encourages a programming style that ignores the differences between local and remote access, such as network bandwidth [80], and treats the unavailability of a resource or a failure as an extreme case. But in an environment where tens of thousands of devices and services come and go, change is inherent and the unavailability of some resource is a frequent occurrence.

Second, RPC packages and distributed object systems, such as Legion [67] or Globe [109], compose distributed applications through programmatic interfaces. Just like transparent access to remote resources, composition at the interface level simplifies application development. However, compo-

sition through programmatic interfaces also leads to a tight coupling between major application components because they directly reference and invoke each other. As a result, it is unnecessarily hard to add new behaviors to an application. Extending a component requires interposing on the interfaces it uses, which requires extensive operating system support [54, 87, 101] and is unwieldy for large or complex interfaces. Furthermore, extensions are limited by the degree to which extensibility has been designed into the application's interfaces.

Third, distributed object systems encapsulate both data and functionality within a single abstraction, namely objects. Yet again, encapsulation of data and functionality extends a convenient programming paradigm for single-node applications to distributed systems. However, by encapsulating data behind an object's interface, objects limit how data can be used and complicate the sharing, searching, and filtering of data. In contrast, relational databases define a common data model that is separate from behaviors and thus make it easy to use the same data for different and new applications. Furthermore, objects as an encapsulation mechanism are based on the assumption that data layout changes more frequently than an object's interface, an assumption that may be less valid for a global distributed computing environment. Increasingly, many different applications manipulate the same data formats, such as XML [15]. These data formats are specified by industry groups and standard bodies, such as the World Wide Web Consortium, and evolve at a relatively slow pace. In contrast, application vendors compete on functionality, leading to considerable differences in application interfaces and implementations and a much faster pace of innovation.

Not all distributed systems are based on extensions of single-node programming methodologies. Notably, the World Wide Web does not rely on programmatic interfaces and does not encapsulate data and functionality. It is built on only two basic operations, GET and POST, and the exchange of passive, semi-structured data. In part due to the simplicity of its operations and data model, the World Wide Web has successfully scaled across the globe. Furthermore, the narrowness of its operations and the uniformity of its data model have made it practical to support the World Wide Web across a huge variety of devices and to add new services, such as caching [22, 103], content transformation [36], and content distribution [53].

However, from a pervasive computing perspective the World Wide Web also suffers from three significant limitations. First, just like conventional distributed systems, it places the burden of adapting to change on users, for example, by making them reload a page when a server is unavailable be-

cause, for example, it is overloaded or inaccessible. Second, it requires connected operation for any use other than reading static pages. Finally, it does not seem to accommodate emerging technologies that are clearly useful for building adaptable applications, such as service discovery [2, 5, 25] and mobile code [106]. While Java applets are a form of mobile code, they are only active while the corresponding page is displayed and, by default, can only communicate with the originating server. As a result, they are basically limited to enlivening web pages and implementing site-specific chat clients.

2.1 *The Unique Requirements of Pervasive Computing*

The inadequacy of existing distributed systems raises the question of how to structure system support for pervasive applications. On one side, extending single-node programming models to distributed systems leads to the shortcomings discussed above. On the other side, the World Wide Web avoids several of the shortcomings but is too limited for pervasive computing. To help define a better alternative, we identify the three unique requirements of pervasive computing.

Requirement 1 *Embrace contextual change.*

As people move through the physical world, the execution context of their applications changes all the time. It is impractical to ask users to manually manage these changes, such as entering a new wireless network name and access key every time they enter a different network. Systems thus need to expose contextual changes, rather than hiding distribution, so that applications can implement their own strategies for handling changes and spare the users from doing so. Event-based notification or callbacks are examples of suitable mechanisms. At the same time, systems need to provide primitives that simplify the task of adequately reacting to change. Examples for such primitives include “checkpoint” and “restore” to simplify failure recovery, “move to a remote node” to follow a user as she moves through the physical world, and “find matching resource” to discover suitable resources on the network, such as nearby instruments in a biology laboratory or other users with whom to exchange messages.

Requirement 2 *Encourage ad hoc composition.*

As people use different devices in different locations, they expect that applications and devices just plug together. It is impractical to ask users to manually perform the composition. Systems thus should make it easy to compose applications, services, and devices at runtime. In particular, installing a user's applications on a device must be easy. Furthermore, interposing on an application's interactions with other applications and network services must be simple. Interposition makes it possible to dynamically change the behavior of an application or add new behaviors without changing the application itself. This is particularly useful for complex and reusable behaviors, such as replicating an application's data or deciding when to migrate an application.

Requirement 3 *Recognize sharing as the default.*

In essence, pervasive computing strives to make information accessible anywhere and anytime. It is impractical to ask users to manage the corresponding files (by, for example, moving them between different devices) and to convert between different data formats. Systems thus need to make it easy to access saved information and to share information between different applications and devices. Ease of sharing is especially important for services that search and filter large amounts of data. At the same time, data and functionality depend on each other, for example, when migrating an application and its data. Systems thus need to include the ability to group data and functionality but must make them accessible independently.

Individually, the three requirements have been recognized by previous distributed systems efforts. For instance, Bayou [88, 102] exposes different data values in the data stores on different devices, and Odyssey [81] relies on asynchronous notifications to expose contextual change to applications. Furthermore, as already discussed, the World Wide Web is built on only two basic operations, GET and POST, which simplifies dynamic composition, and the recent move towards expressing all data on the Internet as XML attempts to facilitate sharing. Our approach differs from these efforts in that we advocate addressing all three requirements at the same time.

Common to the requirements is the realization, similar to that behind extensible operating systems [9, 29, 57], that systems cannot automatically decide how to react to change, because there are too many alternatives. Where needed, the applications themselves should be able to determine and implement their own policies [94]. As a result, we are advocating a structure different from previous

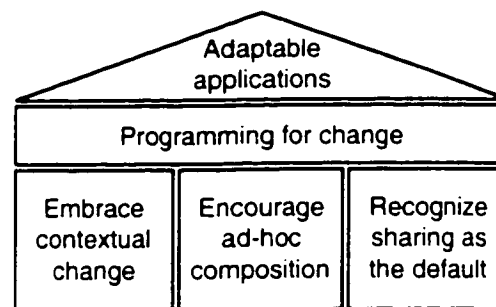


Figure 2.1: Illustration of our approach. The three requirements guide the design of our system architecture and make it feasible for application developers to program for change, resulting in adaptable applications.

distributed systems, which exposes distribution so that applications can adapt to change instead of users.

At the same time, the three requirements do not preclude the use of established programming methodologies. Embracing contextual change does not prevent us from providing reasonable default behaviors. But it does emphasize that applications must be notified of change. Similarly, encouraging ad hoc composition does not preclude the use of strongly typed APIs. However, it does emphasize the need for simplifying interposition. Finally, recognizing sharing as the default does not preclude the use of object-oriented programming. The ability to abstract data or functionality is clearly useful for structuring and implementing applications. At the same time, ease of sharing, with its emphasis on the ability to search, filter, and translate data, does suggest that application data and functionality build on distinct abstractions.

More importantly, a system architecture whose design meets the three requirements provides considerable support for coping with change. Embracing contextual change helps with identifying and reacting to changes in devices and the network. Encouraging ad hoc composition helps with changes in application features and behaviors. Finally, recognizing sharing as the default helps with changes in data formats and the corresponding application functionality. Given a system that meets these requirements, application developers can focus on making applications adaptable, and the users, in turn, can focus on their tasks instead of manually adapting their devices and applications. While programming adaptable applications requires developer discipline when compared to conven-

tional distributed systems, it also provides an extraordinary opportunity to transform the way people interact with their computers and applications. This approach to building pervasive applications is illustrated in Figure 2.1.

2.2 Adaptability and Transparency

By exposing distribution, our approach differs from a large class of efforts that have explored how to build services that adapt (largely) transparently to an ever changing execution context. Since an exhaustive survey of such efforts is beyond the scope of this dissertation, we highlight a few, representative storage and networking services and discuss their relationship to our approach.

In the area of adaptable storage services, the Coda file system [58, 78] aggressively caches files on clients—hoarding files even before they might be accessed—to support disconnected and weakly connected operation. Similarly, the Rover system [55] caches service objects on clients and provides queued RPC to support mobile devices that may only be intermittently connected. The University of California at Berkeley’s xFS file system [4] automatically distributes file storage, caching, and control across a set of cooperating workstations and thus eliminates the need for dedicated file servers. Finally, the OceanStore project [59] is trying to create a global object store that runs on an untrusted computing infrastructure and automatically moves and replicates data between devices to optimize for locality and availability.

In the area of adaptable networking services, the Mobile IP architecture [51] supports device mobility by automatically forwarding TCP/IP traffic, even if a device is not connected to its home network. Furthermore, the Barwan networking architecture [16] includes support for transparently switching between wired and wireless networks and for correspondingly adjusting the data transport protocols to ensure that devices remain continuously and reliably connected, independent of their current location.

Common to these efforts is that they provide particular *services*, rather than representing programming methodologies like the RPC and distributed object systems discussed above. As a result, they can focus on making the provided services adaptable. More fundamentally, to be as transparent as possible, these efforts also share a drive to contain changes to existing applications and networking infrastructure as much as possible. One important technique used by many of these

efforts is the use of proxies. For example, xFS includes NFS proxies that provide access to its file system data to unmodified Unix-based client machines. Furthermore, Rover includes a web browser proxy to provide web access to mobile devices while also leaving existing browsers and servers unchanged. Finally, both Mobile IP and Barwan use proxies to isolate protocol additions to the mobile clients and their routers, with Barwan also generalizing proxies under their TACC (for transformation, aggregation, caching, and customization) model [36]. As a result, these services can adapt to a changing execution context, while also being able to transparently interact with legacy systems and applications.

Overall, we believe that these efforts are complimentary to our own approach along two axes. First, by providing continuous access to important services, such as storage and networking, these efforts certainly lessen the burden of making applications adaptable and thus simplify the development of pervasive applications. In fact, *one.world* also reflects the desire to isolate applications from at least some changes: As discussed in Chapter 5.3, the implementation of our architecture's discovery service relies on an automatically elected discovery server and thus transparently adapts to a changing device and network topology. Discovery server elections ensure that the directory of discoverable resources is almost always available while also hiding the directory's location. This implementation trade-off is reasonable, as directory availability is considerably more important to pervasive applications than directory location.

Second, since a system architecture that meets the three requirements has been specifically designed for implementing adaptable programs, we believe that such an architecture can also simplify the implementation of transparently adapting services—an important concern when considering the complexity of services such as xFS or OceanStore. Furthermore, several of the above systems are only transparent to a degree and need to expose some changes to applications. For example, the resolution of file conflicts in Coda is at least type-specific if not application-specific [60]. Furthermore, Barwan needs to notify applications that the currently used networking technology has changed so that they can adapt, for example, the fidelity of streaming audio or video to better match available bandwidth. Clearly, a system architecture that follows our approach provides a convenient framework for exposing these changes.

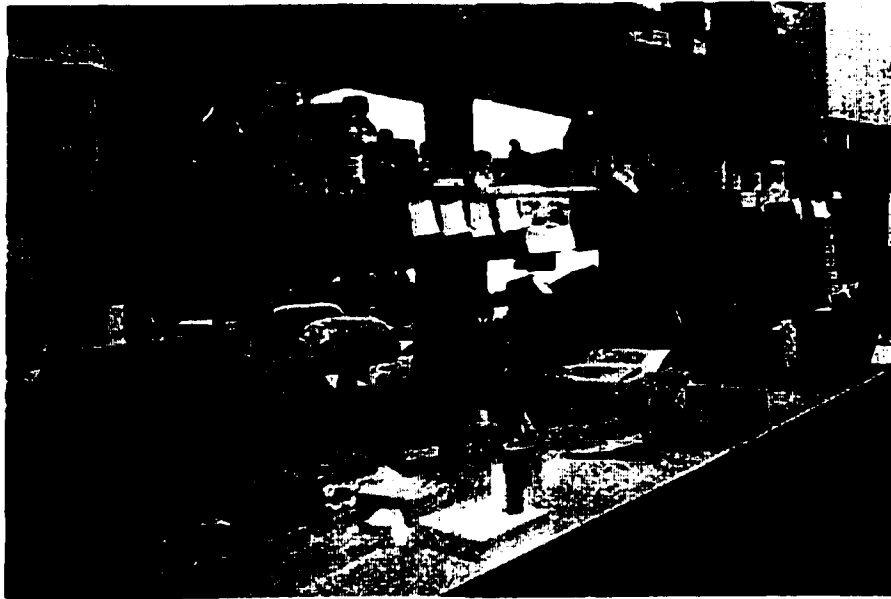


Figure 2.2: A workbench in a biology laboratory. Notice how the touchscreen on the right hand side becomes just another instrument on this workbench, rather than being the focus of attention.

2.3 The Biology Laboratory as an Example Application Domain

To illustrate the three requirements central to our approach to building pervasive applications, we now introduce the digital biology laboratory. Unlike the scenarios presented in [31] and [113], the digital laboratory does not illustrate the full potential of pervasive computing. However, it addresses a real need of real people—performing reproducible biology experiments. Furthermore, as discussed in Chapter 6.4.2, the digital laboratory has been implemented on top of *one.world* by the University of Washington’s Labscape project and has been deployed at the Cell Systems Initiative [6]. As a result, it provides an apt example for a pervasive application and the three requirements.

As already mentioned, the goal of researchers working in a biology laboratory is to perform reproducible experiments. Today, they manually log individual steps in their paper notebooks. This easily leads to incomplete experimental records and makes it unnecessarily hard to share data with other researchers, as the biologists need to explicitly enter the data into their PCs. In contrast, a digital laboratory employs digitized instruments, such as pipettes and incubators, to automatically

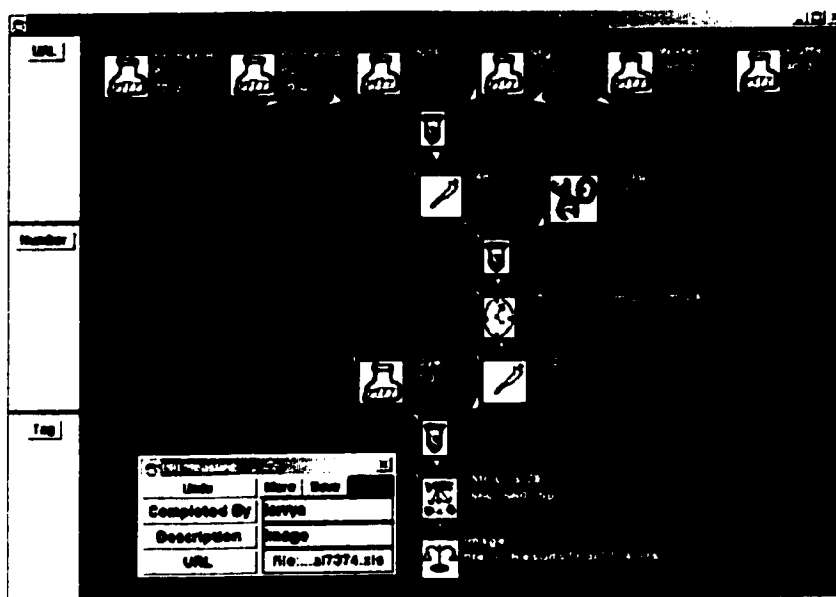


Figure 2.3: A screenshot of Labscape's user interface. Each experiment is represented as an experimental flowgraph, or *guide*. The individual icons represent different experimental steps and the arrows represent ordering constraints. Originally, the guide functions as a plan for the experiment to be performed. As the researcher performs a step, she annotates the corresponding icon with the results of that step. Over time, the guide thus becomes a record of the experiment. Note that this user interface was developed by the Labscape project through user interface studies with actual biology researchers.

capture data, location sensors to track researchers' movements, and touchscreens to display experimental data close to the researchers (see Figure 2.2 for a workbench in the digital laboratory). As a result, biologists in the digital laboratory have more complete records of their experiments and can more easily share results with their colleagues.

A fundamental feature of the digital laboratory is that experimental data follows a researcher as she moves through the laboratory. Furthermore, the data can follow her as she leaves the laboratory, for example, so that she can review a day's results on her tablet computer while taking the commuter train home. At the same time, there is no need to move the entire digital laboratory application as the researcher moves through the physical world. Rather, only a small component to capture and display experimental data needs to follow the researcher. Eventually, all data is forwarded to a centralized repository, making it possible, for example, to mine the data of several experiments.

Figure 2.3 shows a snapshot of the capture and display component's user interface, which is also called a *guide* and has been developed by the Labscape project through user interface studies with actual biology researchers. Each experiment is represented as a flowgraph. The individual icons represent different experimental steps and the arrows represent ordering constraints between the steps. Originally, the guide functions as a plan for the experiment to be performed. A researcher can either select a flowgraph from a library of existing flowgraphs or create her own (which may be based on an existing flowgraph). As the researcher performs a step, she annotates the corresponding icon with the results of that step, or, if she is using digital instruments, they are automatically annotated. Over time, the guide becomes a record of the experiment.

The three unique requirements of pervasive computing—change, ad hoc composition, and pervasive sharing—show up in the digital laboratory as follows:

Embrace contextual change. Biology laboratories are organized into task-specific workareas, often centered around a specific instrument, such as a centrifuge or incubator, and biologists often move between workareas while working on the same experiment. As a result, a researcher's location changes needs to be exposed to her guide, so that it can automatically follow her to the corresponding touchscreens.

Encourage ad hoc composition. As the researcher moves through the biology laboratory, her guide needs to transparently connect to the digital instruments in her current workarea and incorporate readings into the experimental flowgraph. Furthermore, as outside researchers visit the laboratory, their PDAs or laptops need to be automatically integrated into the digital laboratory application so that the researchers can exchange and review experimental results.

Recognize sharing as the default. Biology experiments often last hours, if not days, and biologists multitask between several experiments at the same time or focus on performing a single step for many experiments in a row. As a result, it must be easy to switch between different experiments, to hand off experiments between researchers, and to compare different experiments.

As a pervasive application, the digital laboratory application needs to be considerably more seamless and adaptable than conventional distributed applications. At the same time, it also illustrates an important property of pervasive computing environments: their human scale. In particular,

the digital laboratory only needs to scale to a limited number of concurrent users, as only so many people can work in the same laboratory at the same time. Furthermore, the digital laboratory application typically needs to adapt at a human time scale. A researcher walking from one workarea to another thus leaves a relatively large timespan (compared to the microsecond latencies often considered in distributed systems work) for migrating the researcher's guide and connecting to close-by instruments.

2.3.1 *Short-comings of the Status Quo*

To emphasize that pervasive applications are not just conventional distributed applications, we now consider the limitations of conventional systems in implementing the digital laboratory. First, it is hard to move between devices. Even with existing networked application support, such as X Windows [82] or roaming profiles in Windows [108], users have to manually log into a machine, start their applications, and load the necessary data. Second, it is hard to connect to different instruments as researchers move between workareas. Conventional systems focus on providing point-to-point communications (with TCP being the most prominent example) and lack facilities for dynamically discovering and connecting to close-by instruments without explicit, manual configuration. Finally, it is hard to share data. On one hand, file systems support only coarse-grained sharing—remember that biology experiments consist of many steps that typically add only a small amount of data to an experiment's record. On the other hand, databases are difficult to set up and administer, typically requiring dedicated facilities and staff.

Based on similar observations, several efforts have explored how to layer additional middleware for building adaptable applications onto existing distributed systems. Sun's Jini is probably the most popular example for such a middleware platform [5]. Like *one.world*, it supports distributed events, tuple storage, and service discovery. However, unlike *one.world*, it is layered on top of Java RMI [100], a traditional distributed object system, and thus inherits RMI's limitations. In particular, Jini requires a statically configured infrastructure to run its discovery server. Furthermore, it requires an overall well-behaved computing environment because it relies on transparent and synchronous remote invocations, does not provide isolation between applications running within the same Java virtual machine, and links objects on different devices with each other through distributed garbage

collection. In other words, Jini is inherently limited because it builds on a conventional distributed system. To be effective, system support for pervasive applications must be designed from the ground up to meet the three requirements of change, ad hoc composition, and pervasive sharing. We discuss the differences between *one.world* and Jini and the latter's limitations in more detail in Chapter 8.

Chapter 3

ARCHITECTURE

With the three requirements in place, we now introduce *one.world* and its services. Our architecture is structured according to the following four principles. First, bias a set of foundation services to directly address the three requirements of change, ad hoc composition, and pervasive sharing. Second, express specific system services in terms of the foundation services and make them available as common application building blocks. Third, employ a classic user/kernel split, with foundation and system services provided by the kernel, and libraries, system utilities, and applications running in user space. Finally, remain neutral on other issues, such as whether to implement applications as client/server or peer-to-peer applications. The resulting organization is illustrated in Figure 3.1. We now present the individual services as well as the provided library support in more detail.

3.1 Foundation Services

The four foundation services directly address the three requirements of change, ad hoc composition, and pervasive sharing. First, a *virtual machine*, such as the Java virtual machine [70] or Microsoft's common language runtime [104], provides a common execution environment across all devices and hardware platforms.¹ Since developers cannot possibly predict all the devices their applications will run on, the virtual machine ensures that applications and devices are composable. Second, *tuples* define a common data model, including a type system, for all applications and thus make it easy to share data. They are records with named fields and are self-describing in that an application can dynamically determine a tuple's fields and their types. Third, all communications in *one.world*, whether local or remote, are through *asynchronous events*; applications are composed from components that exchange events through imported and exported event handlers. Events make change explicit to applications, with the goal that applications adapt to change instead of forcing users to

¹*one.world* is implemented in Java. Though, its implementation does not rely on any features that are unique to Java, and it could be implemented on a different virtual machine platform, such as Microsoft's common language runtime.

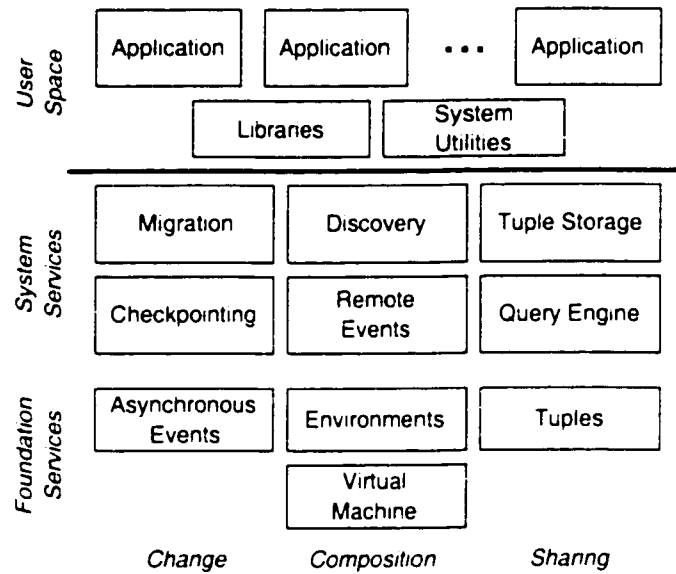


Figure 3.1: Overview of *one.world*'s architecture. Foundation and system services are part of the kernel, while libraries, system utilities, and applications run in user space.

manually reconfigure their devices and applications.

Finally, *environments* are the central mechanism for structuring and composing applications. They serve as containers for stored tuples, application components, and other environments and form a hierarchy with a single root per device. Each application consists of at least one environment, in which it runs and stores its persistent data. However, applications are not limited to a single environment and may span several, nested environments. Comparable to processes in conventional operating systems, environments provide protection and isolate applications from each other and from *one.world*'s kernel, which is hosted by each device's root environment. Environments also are an important mechanism for dynamic composition: an environment controls all nested environments and can interpose on their interactions with the kernel and the outside world. Environments thus represent a combination of the roles served by file system directories and nested processes [17, 33, 107] in other operating systems. Figure 3.2 shows an example environment hierarchy.

To reiterate, the design rationale for the foundation services is as following. First, a virtual machine supports ad hoc composition between applications and devices. Second, tuples define a

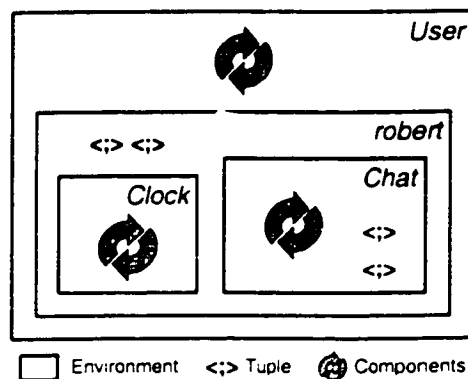


Figure 3.2: Illustration of an example environment hierarchy. The *User* environment hosts the Emcee application and has one child, named *robert*, which stores several tuples representing that user's preferences. The *robert* environment in turn has two children, named *Clock* and *Chat*. The *Clock* environment only contains active application components, while the *Chat* environment, in addition to hosting the Chat application, also stores tuples representing the music being broadcast by Chat.

Table 3.1: Application needs and corresponding system services.

Applications need to...	<i>one.world</i> provides...
Search	Query engine
Store data	Structured I/O
Communicate	Remote events
Locate	Discovery
Fault-protect	Checkpointing
Move	Migration

common type system for all applications and thus simplify the sharing of data. Third, events make change explicit to applications, so that they adapt to change instead of users. Finally, environments host application components, store persistent data, and—through nesting—facilitate the composition of applications and services.

3.2 System Services

In addition to the foundation services, *one.world* provides a set of system services that serve as common application building blocks. Table 3.1 summarizes common application needs and the

corresponding system services.

The *query engine* provides the ability to search tuples by instantiating filters. Queries support comparison of a constant to the value of a field, comparison to the type of a tuple or field, and negation, disjunction, and conjunction. *Structured I/O* lets applications access stored tuples in environments. It supports the writing, reading, querying, and deleting of tuples. The structured I/O operations are atomic so that their effects are predictable, which is especially important when several applications concurrently access tuples in the same environment, and can optionally use transactions to group several operations into one atomic unit. The query engine and structured I/O simplify data access because applications can directly access relevant data items.

Remote event passing (REP) forwards events to remote services and is *one.world*'s basic mechanism for communicating across the network. Consistent with our push towards exposing distribution and in contrast to RPC or distributed object systems, remote communications in *one.world* are explicit. To use REP, services export event handlers under symbolic descriptors, that is, tuples, and clients send events by specifying the symbolic receiver. *Discovery* locates services with unknown locations. It supports a rich set of options, including early and late binding [2] as well as anycast and multicast, and is fully integrated with REP, resulting in a simple, yet powerful API. Discovery is especially useful for applications that migrate or run on mobile devices and need to find local resources, such as a close-by digital instrument.

Checkpointing captures the execution state of an environment tree and saves it as a tuple, making it possible to later revert the environment tree's execution state. Checkpointing simplifies the task of gracefully resuming an application after it has been dormant or after a failure, such as a device's batteries running out. *Migration* provides the ability to move or copy an environment and its contents, including stored tuples, application components, and nested environments, either locally or to another device. It is especially useful for applications that follow a person from shared device to shared device as she moves through the physical world.

3.3 Library Support

Outside of *one.world*'s kernel, our architecture provides additional, user-level library support for implementing pervasive applications. The libraries include functionality for constructing an appli-

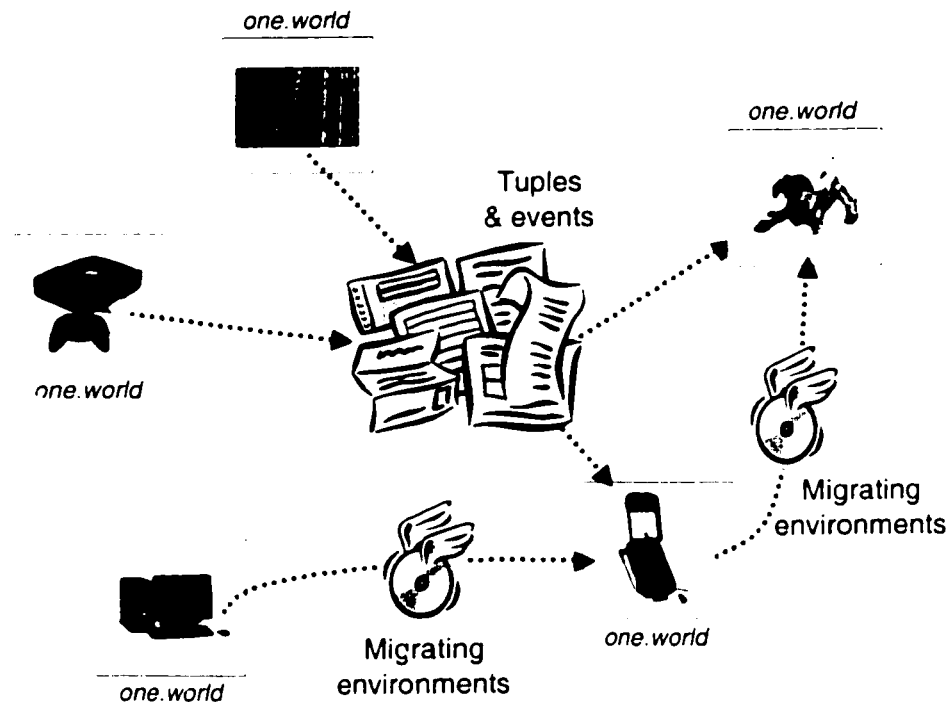


Figure 3.3: The big picture. All the different devices in a pervasive computing environment run the same system platform, *one.world*. The different applications exchange tuples and events between each other and migrate from one device to another.

cation's user interface and for the timed execution of event handlers. More importantly, *operations* help manage asynchronous interactions. They are based on what we call the logic/operation pattern. This pattern structures applications into logic—computations that do not fail, such as creating and filling in a message—and operations—interactions that may fail, such as sending the message to its intended recipients. Operations simplify such interactions by keeping the state associated with event exchanges and by providing automatic timeouts and retries.

3.4 The Big Picture

Pulling back, Figure 3.3 illustrates the big picture behind our architecture. The basic idea is that all the different devices in a pervasive computing environment run the same system platform, namely *one.world*. While individual devices may provide additional services over to those supplied by

our architecture, applications can rely on the same basic operating environment on every device (modulo differences in CPU speed and memory/storage capacity). Each device is independent of other devices and need not be connected with (all) other devices. Furthermore, each device may be administered separately.

The different applications running on the different devices interact with each other by exchanging tuples and events. When communicating with other applications, an application may not necessarily know how many applications it is communicating with and where those applications are located (hence the heap of data in the figure). Furthermore, as people move through the physical world, applications follow by migrating from device to device to device.

For example, in case of the digital biology laboratory, the digital instruments send events describing performed operations to a researcher's guide, and the guide, in turn, forwards experimental results to the centralized data repository. As the researcher is switching between work areas, her guide follows her by migrating from touchscreen to touchscreen to touchscreen. Similarly, when using Emcee and Chat, a user's Chat application sends text and audio messages by sending events to other people's instances of Chat. As the user moves between rooms, Emcee migrates her Chat application so that Chat is always running on a device close to that person.

3.5 *one.world and Distributed Systems Concerns*

Like any distributed system, *one.world* must address several distributed systems issues, such as how to provide processes, storage, and communications. Table 3.2 lists the most important issues and relates them to the corresponding features in our architecture. When compared to Figure 3.1 on page 21, it represents an alternative organization of our architecture. The table also lists the specific chapters that discuss these features in detail, thus serving as an index into the programming model and implementation chapters of this dissertation. The *request/monitor mechanism* referenced in the table is the interposition mechanism enabled by nested environments: it lets an outer environment interpose on all interactions of nested environments with the kernel and the outside world.

Table 3.2: *one.world* and distributed systems concerns. *Issue* specifies the distributed systems concern. *Feature* describes the corresponding *one.world* service. *Chapter* lists the programming model or implementation chapter discussing that feature.

Issue	Feature	Chapter
Process management	Environments contain applications.	4.1
	Checkpointing captures an application's execution state and migration moves or copies an application.	4.5, 5.4
Addressability / Naming	Protection domains limit access to direct references.	4.1, 5.1
	The environment hierarchy limits access to nested environments.	4.1
	Discovery and remote event passing provide access to arbitrary event handlers.	4.4, 5.3
Execution model	Applications are composed from components that exchange asynchronous events.	4.3, 5.1
	Operations manage event exchanges, notably those with event handlers outside an application.	4.3.2, 5.2
Storage	Structured I/O persistently stores tuples in environments and the query engine performs searches across stored tuples.	4.2
Communications	Discovery locates remote receivers and remote event passing sends events to remote receivers.	4.4, 5.3
Security	Protection domains isolate applications.	4.1, 5.1
	The request/monitor mechanism can be used to implement reference monitors and auditing.	4.3.1
Resource allocation	The request/monitor mechanism can be used to interpose on requests for system services.	4.3.1
Extensibility	The request/monitor mechanism can be used to add new services.	4.3.1

Chapter 4

PROGRAMMING MODEL

We now explore *one.world*'s programming model in detail. We describe our architecture's services and their operations, give code examples, and explain our design decisions. In contrast to the previous chapter, which is organized around horizontal slices through our architecture—foundation services, system services, and library support, this chapter is organized around conventional systems concerns, focusing on the perspective of application developers becoming familiar with our architecture. In particular, we start with namespaces in Chapter 4.1 and introduce environments as containers for applications, their persistent data, and other environments. We follow with data management in Chapter 4.2 and discuss tuples, the query language and query engine, as well as structured I/O. In Chapter 4.3, we explore *one.world*'s execution model and present events, the request/monitor mechanism, and operations. Next, in Chapter 4.4, we discuss the communication model and describe discovery and remote event passing. Finally, in Chapter 4.5, we explore application persistence and present checkpointing and migration. We do not further discuss virtual machines, as our architecture directly builds on existing virtual machine technology.

In summary, an application in *one.world* consists of an environment, which acts as the namespace for the application's objects and can include code, data, and other environments. The application's code executes in response to asynchronous events, which may be generated by the system, the application itself, or by other applications, potentially on other devices. Events are delivered through a rich event delivery interface. The application's data is stored in an associative tuple store, which is part of the application's environment. The application's execution state can be checkpointed and stored in its environment so that the application can later be reverted to the saved state. The application can also be migrated to a different device, which either moves the application's environment and all its contents to that device or creates a copy on the remote device. *one.world* itself executes as a set of kernel services, available to, and mediating, all applications running on a device.

4.1 Namespaces

Comparable to processes in conventional operating systems, environments host applications and—to ensure that different applications cannot directly interfere with each other and can be managed independently—isolate them from each other through protection domains. By default, each environment represents its own protection domain; though, a protection domain may span several, nested environments. To enforce isolation, all data is copied between protection domains. Applications can only exchange event handlers, thus enabling them to communicate with each other. Furthermore, operations on environments and access to an environment's tuple storage are limited to the requesting environment and its descendants, thus making it possible to limit an application's effects to its subtree, which is important for pervasive computing environments as potentially untrusted applications move from one device to another.

Environments provide structure not only by isolating applications from each other, but also by grouping application functionality and persistent data within the same container. The grouping of functionality and data enables *one.world* to load an application's code from its environment and to store the application's checkpoints with the application. More importantly, it simplifies the development of pervasive applications that follow a user through the physical world, as an application and its data, including code and checkpoints, can be migrated in a single operation. At the same time, environments always maintain a clear separation between functionality and data, which can be accessed independently and, unlike objects, are not hidden behind a unifying interface. In addition to providing structure, environments provide control through nesting: an outer environment has full control over an inner environment, including the ability to interpose on the inner environment's interactions with the kernel and the outside world. Nesting thus makes it possible to easily factor important pervasive computing features, such as the logic to control migration and the ability to synchronize data with other devices, out of an application and reusing that functionality across several applications. To exploit nesting for this purpose, the reusable functionality is provided by an outer environment, and the application relying on that functionality is placed into an inner environment.

Table 4.1 summarizes the environment operations. The majority of these operations works as expected and is used to create and delete environments and to start and stop applications. The *checkpoint*, *restore*, *move*, and *copy* operations are used for checkpointing and migration and are

Table 4.1: The environment operations. *Operation* specifies the environment operation, and *Explanation* describes the operation.

Operation	Explanation
<i>create</i>	Create a new environment.
<i>rename</i>	Rename an environment.
<i>load</i>	Load an application into an environment.
<i>activate</i>	Activate the application.
<i>terminate</i>	Terminate the application.
<i>unload</i>	Unload the application.
<i>destroy</i>	Delete the environment and all its contents.
<i>move</i>	Move the environment and all its contents.
<i>copy</i>	Copy the environment and all its contents.
<i>checkpoint</i>	Checkpoint the environment.
<i>restore</i>	Restore a previously captured checkpoint.

described in detail in Chapter 4.5. To perform an operation, an application specifies the operation, the targeted environment, and any additional arguments as necessary. Environments are named by either a globally unique identifier [65] (GUID) or a human-readable path name, which, like a path name in Unix, is composed of individual environments' names separated by slashes ('/'). An environment's GUID cannot be changed after creation, so that it can be used as a unique reference for that environment. In contrast, an environment's human-readable name, just like a directory name in conventional operating systems, can be changed after creation through the *rename* operation to accommodate changing user needs. *one.world*'s kernel runs in a device's root environment, which, just like the root directory in Unix, is named "/".

4.2 Data Management

Data management in *one.world*, that is, the ability to query, store, and exchange information, is based on tuples. Tuples define the common data model, including the type system, for applications running in our architecture. They are self-describing, mutable records with named and (usually) typed fields. Valid field types include numbers, strings, and arrays of basic types, as well as tuples, thus allowing tuples to be nested within each other. Arbitrary objects can be stored in a tuple in form of a special container that encapsulates a serialized representation of the object.

By providing a common, structured data model, tuples enable our architecture's data management services, notably the query engine and structured I/O. As a result, tuples let pervasive applications directly encode and exchange the information they manage. They also obviate the need for separate internal and external representations and for translating between different data formats, generally simplifying the sharing of information. Consider, for example, a personal information management application. It can directly encode a user's appointments, contacts, notes, and messages as tuples and, through the query engine and structured I/O described below, search, store, and exchange that data. As a result, it becomes easier to make a user's data available throughout her living and working spaces and to synchronize between different devices, applications, and people.

To capture the structure of application data, tuples are statically declared and strongly typed. They have a fixed set of fields with specific types and the overall tuple has a type. However, our architecture also includes a special tuple, called `DynamicTuple`. In contrast to other tuples, the fields of a dynamic tuple can be dynamically added and removed and are dynamically typed, that is, they can have any allowable field type. As a result, dynamic tuples are more flexible, but do not offer typing guarantees. They are useful for representing ad hoc data, such as another tuple's metadata, or for prototyping data records during application development.

All tuples share the same base class and have an ID field specifying a GUID to support symbolic references, as well as a metadata field to support application-specific annotations. Each tuple also has a set of methods to programmatically reflect its structure and to access its data, thus allowing applications to inspect and access data items with unknown types. Finally, each tuple has methods to validate its semantic constraints (for example, to determine whether a tuple's field values are consistent with each other) and to produce a human-readable representation. The base class for all tuples, which defines these common fields and methods, is shown in Figure 4.1. It illustrates the simple interface for inspecting and accessing tuples programmatically.

4.2.1 Query Language and Query Engine

So that applications can easily search and filter data, such as a user's appointments or contacts, our data model also defines a common query language for tuples. That language is used by the APIs to the structured I/O and discovery services. Queries support the comparison of a constant to the

```

public abstract class Tuple {
    // The ID and metadata fields.
    public Guid      id;
    public DynamicTuple metaData;

    // Programmatic access to a tuple's fields.
    public final Object get(String name) {...}
    public final void   set(String name, Object value) {...}
    public final Object remove(String name) {...}
    public final List   fields() {...}
    public final Class  getType(String name) {...}

    // Validation of a tuple's constraints.
    public void validate() throws TupleException {...}

    // A tuple's human-readable representation.
    public String toString() {...}
}

```

Figure 4.1: Definition of a tuple. All tuples inherit this base class and have an ID field to support symbolic references and a metadata field to support application-specific annotations. They also have a set of methods to programmatically access a tuple's fields, to validate a tuple's semantic constraints, and to convert the tuple into a human-readable representation. A `Guid` is a globally unique identifier. A `DynamicTuple` is a special tuple; its fields can be of any type and, unlike those of other tuples, can be dynamically added and removed. The accessor methods are final and are implemented using reflection. In contrast, individual tuple classes can override the `validate()` and `toString()` methods to define their own semantic constraints and human-readable representation, respectively. Note that the `remove()` method works only for dynamic tuples.

value of a field, including the fields of nested tuples, the comparison of a type to the declared or actual type of a tuple or field, and negation, disjunction, and conjunction. Since queries are data themselves, they are also expressed as tuples.

An example query in our architecture's query language is shown in Figure 4.8 on page 45. It consists of several, nested Query tuples, which express a type comparison (as indicated by the `Query.COMPARE_HAS_SUBTYPE` constant), a value comparison (as indicated by the `Query.COMPARE_EQUAL` constant), and a conjunction (as indicated by the `Query.BINARY_AND` constant). The overall query matches tuples of type `UserDescriptor` whose `user` field equals

Table 4.2: The structured I/O operations. *Operation* specifies the structured I/O operation. *Argument* specifies how tuples are selected for that operation. *Explanation* describes the operation.

Operation	Argument	Explanation
<i>put</i>	Tuple	Write the specified tuple.
<i>read</i>	Query	Read a single tuple matching the specified query.
<i>query</i>	Query	Read all tuples matching the specified query.
<i>listen</i>	Query	Observe all tuples that match the specified query as they are written.
<i>delete</i>	ID	Delete the tuple with the specified ID.

the value of the `fetchUser` variable.

The query engine processes queries over tuples, as expressed in our architecture’s query language. To use the query engine, services and applications instantiate a filter for a specific query (such as the one illustrated above) and then feed tuples to the filter. Tuples matching the query are passed through and tuples not matching the query are dropped.

4.2.2 Structured I/O

Structured I/O builds on our architecture’s data model and lets applications persistently store tuples in environments. Each environment’s tuple storage is separate from that of other environments. Comparable to the primary key in a relational database table, a tuple’s ID uniquely identifies the tuple stored within an environment. In other words, at most one tuple with a given ID can be stored in a given environment. The structured I/O operations support the writing, reading, and deleting of tuples and are summarized in Table 4.2. They are atomic so that their effects are predictable and can optionally use transactions to group several operations into one atomic unit. To use structured I/O, applications *bind* to tuple storage and then perform operations on the bound resource. All bindings are controlled by leases [40], which limit the time an application can access an environment’s tuple storage. Applications can renew these leases to increase the length of access or cancel them to relinquish access.

In the spirit of Unix’s unified interface to storage and networking [73], structured I/O also provides the same basic API for reading and writing tuples across the network. Because standard communication protocols, such as TCP, provide no persistence and employ only limited buffering,

structured I/O networking supports only a subset of the operations shown in Table 4.2. In particular, it only supports the *put*, *read*, and *listen* operations and not transactions. To use structured I/O networking, applications bind to network endpoints instead of tuple storage. Network endpoints can be either UDP or TCP unicast sockets or UDP multicast sockets. Just as bindings for tuple storage, bindings for network endpoints are leased.

We chose to base I/O on a structured data model instead of using unstructured bytestrings because, by definition, tuples preserve the structure of application data and thus simplify the sharing and searching of data. Furthermore, tuples free applications from explicitly marshaling and unmarshaling data during I/O and from implementing their own, internal database functionality, which is a common strategy for desktop applications [74] and leads to considerable duplication of effort between applications from different vendors. We chose tuples instead of XML [15] because tuples are simpler and easier to use. The structure of XML-based data is less constrained and also more complicated, including tags, attributes, and name spaces. Furthermore, interfaces to access XML-based data, such as DOM [64], are relatively complex.

Structured I/O distinguishes between storage and networking, instead of providing a unified tuple space service [21, 26, 37, 79, 116], because such a separation better reflects how pervasive applications store and communicate data. On one hand, many applications need to modify stored data. For example, a personal information manager needs to be able to update stored contacts and appointments. Structured I/O storage lets applications overwrite stored tuples by simply writing a tuple with the same ID as the stored tuple. In contrast, tuple spaces only support the addition of new tuples, but existing tuples cannot be changed. On the other hand, some applications, such as streaming audio and video, need to directly communicate data in a timely fashion. Structured I/O networking provides that functionality. In contrast, tuple spaces store all tuples before delivering them and consequently retain them in storage. This is especially problematic for streaming audio and video, since data tends to be very large. As a result, tuple spaces represent a semantic mismatch for many pervasive applications, providing too little and too much functionality at the same time.

An additional concern is that tuple spaces are not amenable to layering in asynchronous systems. In particular, the *in* or *take* operation—an atomic read and delete—makes it hard to layer additional services, such as replication, on top of a tuple space. The problem is that the tuple to be deleted is only known after the *in* or *take* has been performed by the tuple space service, thus requiring

```
public interface EventHandler {  
    Handle the event.  
    public void handle(Event e);  
}
```

Figure 4.2: The event handler interface. An event handler has a single method that takes the event to be processed as its only argument and returns no result.

that the replication layer intercept both the original request and the corresponding response. In contrast, a replication layer on top of structured I/O only needs to intercept requests, never responses, because requests for the destructive *put* and *delete* operations are sufficiently descriptive to specify the affected tuples.

4.3 Execution Model

Having described *one.world*'s facilities for data management, we now turn to our architecture's execution model. In *one.world*, all functionality is implemented by event handlers that process asynchronous events. Events are appropriate for pervasive applications, as they make changes in an application's execution context—such as a person or device moving to a different location—explicit and thus provide the application with an opportunity to adapt to those changes. Since events are data, they too are represented by tuples. In addition to the ID and metadata fields common to all tuples, all events have a source field referencing an event handler. This event handler receives notification of failure conditions during event delivery and processing, as well as the response for request/response interactions. Furthermore, all events have a closure field, which can be of any allowable tuple field type including a tuple and is declared to be an `Object`. When responding to an event, by sending another event to the original event's source event handler, the closure of the original event is returned with the new event. Closures thus help simplify the implementation of event handlers, as applications can include any additional state needed for processing a response in the closure of the original request.

As shown in Figure 4.2, event handlers implement a uniform interface with a single method

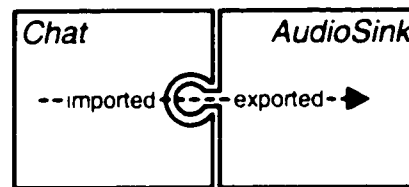


Figure 4.3: Illustration of the relationship between imported and exported event handlers. Boxes represent components, indentations represent imported event handlers, and protrusions represent exported event handlers. The dotted arrow indicates the direction of event flow. In this example, the component named *Chat* imports an event handler named *imported*, and the component named *AudioSink* exports an event handler named *exported*. The two event handlers are linked. When an event is sent to the *imported* event handler, that is, when that event handler is invoked on an event, the event is forwarded to the *exported* event handler, which then processes it. In the case of *Chat* and *AudioSink*, *Chat* sends received audio messages to the *AudioSink*, which then plays back the audio contained in the messages.

that takes the event to be processed as its only argument and returns no result. Any result for a request/response interaction must be sent as a regular event to the event handler referenced by the request's source field. Event delivery has at-most-once semantics, both for local and remote event handling. For remote event handling, at-most-once semantics are appropriate because, in lieu of transactional delivery protocols (which are too heavy-weight for basic event delivery), a remote device may fail after it has accepted an event but before the intended recipient had an opportunity to process it. For local event handling, exactly-once delivery is the norm. However, at-most-once semantics allow *one.world*'s implementation to recover from pathological overload conditions by selectively shedding load.

To simplify code reuse, application functionality is implemented by components. Components are units of code that support a uniform linking protocol and interact solely by exchanging events. They import and export event handlers, exposing the event handlers for linking, and are instantiated within specific environments. Although imported and exported event handlers can be added and removed after component creation, they are typically declared in a component's constructor. Imported and exported event handlers can be linked and unlinked at any time. After linking an imported event handler to an exported event handler, events sent to the imported event handler are processed by the exported event handler. Unlinking breaks this connection again. This relationship between imported

```
public static void init(Environment env, Object closure) {  
    // Create Emcee's component.  
    Emcee comp = new Emcee(env);  
  
    // Link the component with its environment.  
    env.link("main", "main", comp);  
    comp.link("request", "request", env);  
}
```

Figure 4.4: Code example for initializing an application. An initialization method takes as its arguments the environment for the application and a closure, which can be used to pass additional arguments, for example, from a command line shell. The method shown in this figure first instantiates the Emcee component and then links that component with its environment. It links the `main` event handler imported by the environment `env` with the `main` event handler exported by the component `comp`, and the `request` event handler imported by the component `comp` with the `request` event handler exported by the environment `env`. The role of the `main` and `request` event handlers is explained in Chapter 4.3.1. Note that linked event handlers need not have the same name, although they do in this example. This code example is taken from Emcee's source code.

and exported event handlers is illustrated in Figure 4.3.

An application's main component has a static initialization method that instantiates its components and performs the initial linking. It is called by our architecture when loading the application into its environment through the *load* operation listed in Table 4.1. While the application is running, it can instantiate additional components, add and remove imported and exported event handlers, and relink and unlink components as needed. An example initialization method is shown in Figure 4.4. It instantiates a single component, representing Emcee's main component, and then performs two linking operations. After these simple initialization steps, the Emcee application is fully instantiated and ready to execute.

When an event is sent between components in different environments, the invocation of the exported, that is, receiving, event handler on the sent event is performed asynchronously. The corresponding invocation of the imported, that is, sending, event handler returns (almost) immediately. However, when an event is sent between components in the same environment, the event handler invocation is performed as a direct method call, so that the event is delivered reliably and efficiently.

This default can be overridden at link-time, so that events within the same environment are also sent asynchronously.

We chose to use asynchronous events instead of synchronous invocations through, for example, regular procedure calls or a mix between regular procedure calls and asynchronous callbacks for three reasons. First and foremost, asynchronous events provide a natural fit for pervasive computing, as applications often need to raise or react to events, such as sending or receiving a text message or adapting to the execution context after a migration. Second, where threads implicitly store execution state in registers and on stacks, events make the execution state explicit. Systems can thus directly access execution state, which is useful for implementing features such as event prioritization or checkpointing and migration. Finally, taking a cue from other research projects [23, 42, 50, 86, 114] that have successfully used asynchronous events at very different points of the device space, we believe that asynchronous events scale better across different classes of devices than threads.

We chose a uniform event handling interface because it greatly simplifies composition and interposition. Event handlers need to implement only a single method that takes as its sole argument the event to be processed. Events, in turn, have a well-defined structure and are self-describing, making dynamic inspection feasible. As a result, event handlers can easily be composed with each other. For instance, the uniform event handling interface enables a flexible component model, which supports the linking of any imported event handler to any exported event handler.

4.3.1 Interacting with the Kernel and Other Environments

To an application, its environment appears to be a regular component. Each environment imports an event handler called *main*, which must be linked to an application's main component before the application can be started. It is used by *one.world* to notify the application of important events, such as activation, restoration, migration, or termination of the environment, and thus exposes contextual change to the application.

Each environment also exports an event handler called *request* and imports an event handler called *monitor*. Events sent to an environment's request handler are delivered to the first ancestral environment whose monitor handler is linked. The root environment's monitor handler is always linked to *one.world*'s kernel, which processes requests for environment operations, structured I/O,

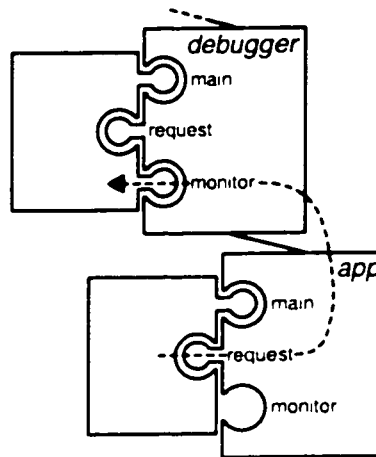


Figure 4.5: Illustration of the request/monitor mechanism. Boxes on the left represent application components and boxes on the right represent environments. The dotted arrow indicates the direction of event flow. The *app* environment is nested within the *debugger* environment. The *debugger* environment's monitor handler is linked and thus intercepts all events sent to the *app* environment's request handler. The use of the main handler is explained in the text.

discovery, and remote event passing. Consequently, the environment request handler provides our architecture's system call interface, and applications use it for interacting with the kernel. For example, Emcee's initialization method as shown in Figure 4.4 on page 36 links to its environment's request handler so that Emcee can utilize *one.world*'s services. Furthermore, by linking to the monitor handler, an application can interpose on all events sent to a descendant's request handler. For example, a debugger can monitor any application simply by nesting the application in its environment, linking to its own monitor handler, and observing the application's request stream. Similarly, a replication service can synchronize any application's data with another device by intercepting the application's structured I/O operations (in fact, as described in Chapter 6.1.1, our replication service does exactly that). This use of the *request/monitor mechanism* is illustrated in Figure 4.5.

As already mentioned in Chapter 4.1, *one.world* enforces the nesting of environments by restricting access to tuple storage and environment operations, such as creating or deleting an environment, to the requesting environment and its descendants. When an application sends an event to its request handler, the event's metadata is tagged with the identity of the requesting environment. Before granting access to tuple storage or performing an operation on an environment, the kernel

verifies that the requesting environment is an ancestor of the environment being operated on.

We chose to represent environments as regular components, because it offers considerable flexibility and power. In particular, the request/monitor mechanism makes interposition trivial and greatly simplifies dynamic composition as illustrated above. Furthermore, because of the uniform event handler interface, the request/monitor mechanism is extensible: it can handle new event types without requiring any changes. Applications can inspect events using the tuple accessor methods shown in Figure 4.1 on page 31, or pass them unexamined up the environment hierarchy. Finally, the same mechanism can be used to provide security by interposing a reference monitor [3] and auditing by logging an application's request stream. It thus obviates the need for fixing a particular security mechanism or policy in *one.world*'s kernel [43].

4.3.2 Reliably Managing Event Exchanges

While asynchronous events provide a good fit for pervasive computing, they also raise the question of how to manage event exchanges, especially when compared to the more familiar thread-based programming model. Of particular concern are how to maintain the state associated with pending request/response interactions and how to detect failures, notably lost events. These issues are especially pressing for pervasive computing environments, where people and devices keep coming and going, and where failures are a common, not an exceptional, occurrence.

Because of our asynchronous execution model, we cannot rely on synchronous invocations, possibly using a thread per invocation, to maintain the state of pending request/response interactions and to detect failures. After all, any response to a request is delivered as a regular event and cannot be returned as the result from the original event handler invocation. Furthermore, as already argued in Chapter 2, we are suspect of a programming model that treats operations that may, and can frequently, fail just like regular procedure or method invocations. It too easily results in applications that do not appropriately account for all failure conditions. At the same time, in our experience with writing event-based code, established styles of event-based programming, such as event loops or state machines, are equally unsuitable. Because they combine all application logic into a single, basic control block, they are only manageable for very simple applications that process few distinct events.

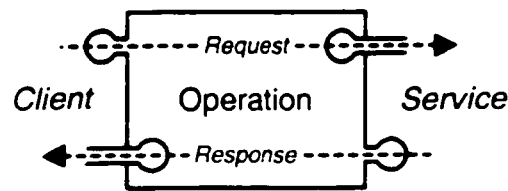


Figure 4.6: Illustration of the operation library. The box represents an operation. Protrusions represent event handlers, which, unlike the event handlers shown in Figure 4.3 on page 35, are directly referenced. The dotted arrows indicate the direction of event flow. To manage asynchronous request/response interactions between a client and a service, an operation is interposed between the event handler accepting requests and the event handler expecting responses. The operation matches each request with exactly one response. It automatically detects timeouts and performs retries.

After some experimentation, we found the following approach, which we call the *logic/operation pattern*, considerably more successful. Under this pattern, an application is partitioned into logic and operations, which are implemented by separate sets of event handlers. Logic are computations that do not fail, barring catastrophic failures, such as creating and filling in a text message. Operations are interactions that may fail, such as sending a text or audio message to its intended recipients. Operations maintain the state associated with these request/response interactions and also include all necessary failure detection and recovery code. A failure condition is reflected to the appropriate logic only if recovery fails repeatedly or the failure condition cannot be recovered from in a general way. As a result, the logic/operation pattern, unlike synchronous invocations, cleanly separates actual application logic from outside interactions, notably I/O, and their failure detection and recovery. Furthermore, unlike event loops or state machines, the logic/operation pattern does allow for the nesting of logic and operations, thus supporting more modular application code and scaling better with application size.

The `Operation` library reifies the logic/operation pattern. As illustrated in Figure 4.6, it is an event handler that connects an event handler accepting requests with an event handler expecting responses. For every request sent to an operation, the operation keeps the state of the pending interaction, including the request's closure, and sends exactly one response to the event handler expecting responses. The operation automatically detects timeouts and performs retries. If all retries fail, it notifies the event handler expecting responses of the failure through an exceptional event.

```
operation = new Operation(0, Constants.OPERATION_TIMEOUT,  
                          timer, request, continuation);
```

Figure 4.7: Code example for creating an operation. The newly created operation does not perform any retries, times out after the default timeout, and uses the timer `timer`. Requests are sent to the `request` event handler and responses are forwarded to the `continuation` event handler. This code example is taken from Emcee's source code.

Operations can be nested and can also be used on both sides of multi-round interactions, such as those found in many network protocols. As a result, operations provide an effective way for expressing complex interactions and structuring event-based applications.

To utilize the operation library, an application simply creates a new operation and then uses the operation instead of the original event handler for issuing requests. Example code for creating an operation is shown in Figure 4.7. The newly created operation connects the `request` event handler for receiving requests to the `continuation` event handler for receiving the corresponding responses. As shown in Figure 4.8 on page 45 and Figure 4.9 on page 50, the operation is then used for issuing requests instead of sending them to the `request` event handler directly.

4.4 Communication Model

To provide ubiquitous information access, pervasive applications need to frequently interact with remote devices and services. As a result, they need to easily locate remote resources and then communicate with them. In this section, we explore how *one.world* addresses these common application needs through service discovery, which provides the ability to locate resources by querying the corresponding resource descriptions, and remote event passing (REP), which provides point-to-point communications between devices. We discuss the two services together because they are not only part of our architecture's communication model but they are also accessed through a single, integrated API. In summary, they are based on a simple model, under which applications name a remote event handler either by a query over a tuple for discovery or by its device and a GUID or name for REP. Discovery resolves the query in a directory that represents all discoverable resources on the

local network, while REP routes events directly to the specified device.

The primary challenge in designing the communications facilities for *one.world* is to provide services that are more flexible than established point-to-point communications technologies and support a rich set of communication patterns. In particular, as people and devices move through the physical world, service discovery assumes a critical role for pervasive applications. After all, if an application cannot locate necessary resources, it cannot function. However, previous discovery systems, such as Jini [5] and INS [2], expose considerably different APIs with distinct options, thus raising the question of what options to support in *one.world*'s discovery service.

To this end, we classify the major discovery options through a set of choices. As illustrated by the following discussion, all the options described below are indeed useful for pervasive applications, and, consequently, they are all supported by *one.world*'s discovery service. The first choice reflects the *binding time* and determines when to perform a discovery query. With early binding, an application first uses discovery to resolve a query and then point-to-point communications to interact with the resolved resource. Early binding is appropriate when an application needs to repeatedly send events to the same resource, such as a specific, close-by wall display, or when services can be expected to remain in the same location, such as those running on dedicated servers. In contrast, late binding [2] combines query resolution and event routing into a single operation, and the discovery service routes the event directly to the matching resource. While late binding introduces a performance overhead for every sent event, it also is the most responsive and thus most reliable form of communication in a highly dynamic environment. The second choice reflects the *specificity* and determines the number of resources receiving an event. Anycast sends the event to a single matching resource, such as the above mentioned wall display, while multicast sends the event to all matching resources, such as all users to chat with.

Taken together, the binding time and specificity cover the design space of previous discovery systems, where services make resources available under descriptors and clients query for matching resources. When determining which of these options to use in an application, the primary choice is whether to rely on early or late binding; whether to use anycast or multicast typically follows directly from the application's requirements. In our experience, late binding is generally preferable over early binding for pervasive applications, as it is more responsive in an ever changing computing environment. However, if an application sends many, possibly large messages to the same receiver

in short succession, the overhead of repeatedly resolving discovery queries becomes too large, and early binding represents the more appropriate choice. At the same time, with early binding, the application needs to be prepared to rediscover the receiver if its computing context changes.

An additional, third choice reflects the *query target* and determines the entity on which to perform a discovery query. Typically, a query is performed on the resource descriptors, and the first two choices assume resource descriptors as query targets. However, the query can also be performed on the events themselves. In this case, an application receives all events sent through late binding discovery that match the query. Using the event as a query target constitutes a form of reverse lookup and is useful for implementing utilities that, for example, log and debug remote communications or bridge between different communication protocols by intercepting messages of one protocol and issuing those of the other. In the former example, the reverse lookups are observing, that is, they do not count as matches for anycasts, while in the latter example, the reverse lookups are consuming, that is, they count as matches for anycasts (after all, the intent is to intercept events).

With our classification of discovery options in place, we now turn to the API for discovery and remote event passing. Both services leverage our architecture's uniform data model and event handling interface to expose a common communications API, which supports all discovery options described above as well as point-to-point communications with only three operations, namely *export*, *resolve*, and *send*. In short, the *export* operation makes an event handler accessible across the network, while the *resolve* operation performs early binding discovery lookups, and the *send* operation routes events both for point-to-point communications and late binding discovery. Conceptually, our architecture's discovery service relies on a directory that represents all discoverable resources on the local network.

In detail, the three operations work as following:

Export. The *export* operation makes an event handler accessible across the network by establishing a mapping between a descriptor and the actual event handler; for discovery, all mappings are collected in a single directory for the local network. The descriptor's type determines how the event handler is exported. If the descriptor is `null` or a `Name`, the event handler is exported for point-to-point communications. If it is a `Query`, the event handler is exported for reverse lookups on the events sent through late binding discovery. For all other tuples, the

Table 4.3: Options for exporting event handlers to remote event passing and discovery. *Descriptor* specifies the tuple under which an event handler is exported. *Explanation* describes how the event handler can be accessed.

Descriptor	Explanation
null	Make the event handler accessible through point-to-point communications. The event handler can be referenced by the GUID returned by the export operation.
Name	Make the event handler accessible through point-to-point communications. The event handler can be referenced by the name contained in the Name tuple.
Query	Make the event handler accessible for reverse discovery lookups. An additional flag specifies whether the reverse lookups are consuming or observing. The former count as matches for anycast, while the latter do not. The event handler cannot be directly referenced. However, events sent through late binding discovery and matching the query are routed to the exported event handler.
All other tuples	Make the event handler accessible for regular discovery lookups. The event handler can be referenced by a query matching the specified tuple.

event handler is exported for regular discovery lookups. Comparable to the use of leases for structured I/O (as described in Chapter 4.2.2), the resulting binding between the event handler and descriptor is leased.

Table 4.3 summarizes the options for exporting event handlers. Note that REP provides two alternatives for exporting event handlers, so that clients can either reference a specific service instance (through a GUID) or a service independent of the current instance (through a name). Furthermore, when exporting an event handler to discovery, the event handler is automatically exported for point-to-point communications as well (as if the descriptor was `null`), so that the discovery service can build on REP for its implementation.

Resolve. The *resolve* operation looks up event handlers in the discovery directory, so that they can be used for point-to-point communications. It takes a query and returns either any or all matching event handlers that have been previously exported for regular discovery lookups. If no event handler matches the query, the *resolve* operation results in a failure notification.

Send. The *send* operation sends an event to a previously exported event handler. The targeted event

```

SymbolicHandler destination;
if (null == fetchLocation) {
    // Location is unknown; use discovery.
    destination = new DiscoveredResource(new
        Query(new Query("",
            Query.COMPARE_HAS_SUBTYPE,
            UserDescriptor.class),
            Query.BINARY_AND,
            new Query("user", Query.COMPARE_EQUAL, fetchUser)));
} else {
    // Location is known; use point-to-point communications.
    destination = new
        NamedResource(fetchLocation, "User " + fetchUser);
}
operation.handle(new RemoteEvent(this, closure, destination, msg));

```

Figure 4.8: Code example for sending a remote event. This example sends the event `msg` for user `fetchUser`, whose location `fetchLocation` may or may not be known. If the location is not known, the event is sent through late binding discovery. The discovery query matches tuples of type `UserDescriptor` whose `user` field equals `fetchUser`. If the location is known, the event is sent through point-to-point communications. The operation forwards the `RemoteEvent` to *one.world*'s kernel, which then performs the actual *send* operation. This code example is taken from Emcee's source code.

handler is specified by a so-called *symbolic handler* that contains the information necessary for routing the event. For late binding discovery, the symbolic handler specifies the query and whether to perform anycast or multicast. The event is delivered to any or all event handlers matching the query in the discovery directory. For point-to-point communications, the symbolic handler specifies the device exporting the event handler and the corresponding GUID or name, and the event is delivered to the event handler that has been exported under the specified GUID or name on the specified device. Both for discovery and point-to-point communications, if no actual event handler matches the symbolic handler, the sender is notified of the failure condition.

Example code for sending an event through both late binding discovery and REP is shown in Figure 4.8. It illustrates how an application can easily switch between either late binding discovery

and point-to-point communications, simply by using a different symbolic handler. Switching from anycast to multicast for late binding discovery is even simpler, as it requires only an additional boolean argument for the constructor of the `DiscoveredResource`.

4.5 Application Persistence

As described in Chapter 4.1, environments are containers for applications, persistent data, and other environments. They provide structure by grouping applications and their data and by isolating different applications from each other. They also provide control by nesting environments within each other and by letting an outer environment interpose on an inner environment's interactions with the kernel and outside world. However, providing structure and control is not enough. In particular, as pervasive applications often run on portable devices, it must be easy to protect them against major failures, such as a device's batteries running out. More importantly and as mentioned several times already, it must be easy to build applications that follow a person from device to device as she moves through the physical world.

To address these important needs of pervasive applications, *one.world* provides checkpointing and migration as common application building blocks. The checkpointing service provides the *checkpoint* and *restore* operations listed in Table 4.1 on page 29. The *checkpoint* operation captures the in-memory state of an environment tree and then stores the captured state as a tuple in the root of the checkpointed tree. The *restore* operation reads a previously stored checkpoint and restores the execution state to the saved state. The migration service provides the *move* and *copy* operations listed in Table 4.1. Both operations capture the in-memory state of an environment tree, move the environment tree, including the just created checkpoint and all stored tuples, to a different device, and then restore the checkpoint. They differ in that the *move* operation deletes the original environment tree, while the *copy* operation leaves the original tree in place. In contrast to transparent migration systems, such as Sprite [28], our architecture's checkpointing and migration services are fully visible to applications. Notably, applications are explicitly notified after they have been restored from a checkpoint or have been migrated to a different device, so that they can adapt to a changed execution context.

As hinted at by this first description, the functionality of the checkpointing and migration ser-

vices can be defined more precisely in terms of three functions: *capture-state()* to create a checkpoint tuple representing an environment tree's in-memory state, *transfer-tree()* to communicate an environment tree's complete contents from one device to another, and *restore-state()* to recreate an environment tree's in-memory state from a previously created checkpoint tuple. Using these three functions, the *checkpoint* operation simply invokes *capture-state()* and stores the resulting checkpoint tuple in the root of the environment tree, while the *restore* operation reads such a checkpoint tuple and then invokes *restore-state()* on the tuple. Both the *move* and *copy* operations represent a sequence of *capture-state()*, *transfer-tree()*, and *restore-state()* invocations, with the difference that the *move* operation also destroys the original environment tree. With this overview over our architecture's checkpointing and migration services in place, we now discuss the three functions in detail.

The *capture-state()* function creates a bytestring representing an environment tree's in-memory state. It relies on the virtual machine to provide a uniform execution platform across different hardware architectures and on object serialization to convert between virtual machine objects and bytestrings. By traversing all objects reachable from a set of well-defined roots—the *main* and *monitor* event handlers introduced in Chapter 4.3.1, the *capture-state()* function captures the in-memory state of the application objects instantiated in the environment tree. Since all communications in *one.world* are through asynchronous events, the *capture-state()* function also captures the environment tree's execution state by serializing pending *(event handler, event)* invocations. Comparable to bus stops in Emerald [98], which define application states that are safe to migrate, execution state can only be captured for pending *(event handler, event)* invocations. Invocations that are currently being executed need to run to completion; invocations that do not complete within a constant waiting period are forcibly terminated. The *capture-state()* function does not capture the state of currently executing *(event handler, event)* invocations, because capturing them requires access to the virtual machine's execution stack. However, many virtual machines, such as the Java virtual machine [70] but unlike the Squeak virtual machine [47], do not explicitly expose their execution stacks and would thus require modifications, which would limit portability.

While the *capture-state()* function does capture the state of the environment tree's application objects and pending *(event handler, event)* invocations, it does *not* include references to resources outside the environment tree. Since environments are isolated from each other, only references to

event handlers can be exchanged between environments; all other data is copied. Consequently, the *capture-state()* function tests each event handler whether it is implemented by code running in one of the environments in the tree. If the event handler is part of the tree, it is written to the checkpoint. If it is not part of the tree, it is replaced by a `null` value. Environments thus provide a well-defined boundary for the state included in a checkpoint, and nulling out event handlers provides a simple contract for revoking access to outside resources. The *capture-state()* function revokes access to outside resources in order to avoid residual dependencies [91], which require an altogether well-connected computing environment. However, pervasive computing environments, with their reliance on wireless networking technologies such as 802.11 [38] or Bluetooth [13], often exhibit weaker connectivity than traditional local networks [78, 102]. Furthermore, disconnected operation is a relatively frequent occurrence, for example, as people travel in cars or on airplanes, and connections, such as those using cell phones, often have high latency and low bandwidth.

The *transfer-tree()* function eagerly communicates an environment tree and all its contents, including the checkpointed in-memory state and all persistently stored tuples, from one device to another in one atomic operation. It is eager, again, because of the weaker connectivity typically found in pervasive computing environments. For a *move* operation, the *transfer-tree()* function invalidates references from the outside into the original environment tree to expose the change in location. When sending an event to such a reference, the event is not transparently redirected through a forwarding address [35]; instead, the sender is notified that the resource has been moved. Invalidating references from the outside into the tree is unnecessary for a *copy* operation, as the original tree remains in place. However, because the original tree remains in place, the *transfer-tree()* function assigns fresh GUIDs to the environments being communicated through a *copy* operation, thus avoiding duplicates.

The *restore-state()* function recreates an environment tree's in-memory state from a checkpoint tuple simply by deserializing it. It then notifies all environments in the tree that they have been restored, moved, or copied. This notification is delivered to each environment's *main* event handler before any other *(event handler, event)* invocation can be performed, which gives the code running in the environment the opportunity to restore access to outside resources before resuming regular event processing. Because the restored environment tree's execution context has likely changed, discovery becomes a central service for reconnecting to outside resources, and, as discussed in Chapter 4.4,

one.world's discovery service has been carefully designed to expose an easy-to-program and flexible interface. Furthermore, we believe that explicitly restoring access to outside resources does not place an additional burden on developers, as applications running on our architecture already need to explicitly acquire resources at other points in their life cycles, such as when they are activated.

One important issue in providing checkpointing and migration is how to control the use of the two services. This issue is especially pressing for migration, as potentially untrusted applications migrate from one device to another, and environment nesting, which gives an outer environment complete control over all nested environments, helps address it. On the sending side, an outer environment can use the request/monitor mechanism to intercept a request to be migrated (that has been issued by a nested environment) and either modify it or reject it. Similarly, on the receiving side, the future outer environments are notified by the *transfer-tree()* function that an environment is about to be migrated to this device, and they can modify the parent environment or reject the migration altogether. Environment nesting thus provides an effective mechanism for limiting how untrusted applications migrate across a network.

Environment nesting also enables an important pattern for initiating checkpointing and migration. Under this pattern, the logic to decide when to checkpoint and restore an application or when and where to migrate an application is factored into its own environment. As a result, the checkpointing or migration logic can be reused across different applications, thus simplifying the development of pervasive applications. In fact, this pattern is used by Emcee, *one.world*'s Finder-like application management utility: As illustrated by the example code in Figure 4.9, it leverages the environment nesting to trivially checkpoint, restore, and move all of a user's applications.

Overall, *one.world*'s checkpointing and migration services leverage our architecture's other services as much as possible to avoid complexity and to provide a clean and useful model for their operation. In particular, they rely on the virtual machine to provide a uniform execution environment across different devices and hardware architectures. They rely on environments to clearly delineate what state to capture and what state not to capture. They also rely on environments for controlling migration, both on the sending and the receiving side, and for factoring the checkpointing and migration logic out of pervasive applications. Furthermore, they rely on the integration of tuple storage with environments to save checkpoints with an application and to migrate an application's data with itself. Next, they rely on asynchronous events to make an application's execution

```
operation.handle(new
    EnvironmentEvent(null, this, EnvironmentEvent.CHECK_POINT,
        env.getId()));
```

```
operation.handle(new RestoreRequest(null, this, env.getId(), -1));
```

```
operation.handle(new
    MoveRequest(null, user, user.env.getId(),
        "sio: "+location+" User", false));
```

Figure 4.9: Code examples for checkpointing, restoring, and moving an environment. The first code snippet checkpoints a user's environment `env`. The second code snippet restores the latest checkpoint for a user's environment `env`. The third code snippet moves a user's environment `user.env` to the device named `location`. For all snippets, the operation forwards the event to *one.world*'s kernel, which then performs the requested environment operation. Note that the first argument to each event's constructor is the source for that event and is automatically filled in by the operation. The code snippets are taken from Emcee's source code.

state explicit. They also rely on asynchronous events to notify the application of a completed *restore*, *move*, or *copy* operation, thus exposing the application's changed execution context. Finally, they rely on discovery so that an application can easily adapt to a changed execution context by restoring access to the appropriate resources. In other words, by building on the other elements of *one.world*'s programming model, our architecture's checkpointing and migration services can provide important functionality not typically found in traditional operating systems.

Chapter 5

IMPLEMENTATION

In this chapter, we present *one.world*'s Java-based implementation. After a short overview, we explore the more interesting and challenging aspects in detail. In particular, we discuss the implementation of environments as application containers in Chapter 5.1, followed by our operation library in Chapter 5.2, discovery and remote event passing in Chapter 5.3, and checkpointing and migration in Chapter 5.4. We summarize the respective implementation challenges and their solutions in Chapter 5.5.

Our implementation of *one.world* is guided by two constraints. First, the implementation should be portable. In particular, it cannot require modifications to the underlying virtual machine, so that we can directly build on existing virtual machines. However, to keep the implementation effort manageable, we do allow for the use of external, platform-native libraries, even though such libraries may reduce portability. Note that, in contrast, all application code must be virtual machine code and cannot access platform-native libraries. Second, the implementation should be stable and efficient enough to support real-world usage in pervasive computing environments with several dozen people and devices—corresponding to shared spaces such as meeting rooms or laboratories. As exemplified by the Labscape digital laboratory application, which we introduced in Chapter 2.3 and discuss in detail in Chapter 6.4.2, pervasive computing environments of this scale are sufficient for gaining actual, everyday user experiences with our architecture. By limiting the targeted scale, we can thus focus our initial efforts on producing a stable and correct implementation, with the expectation that we will revisit scalability issues later on. We present quantitative results, which demonstrate that our implementation meets this constraint, as part of our experimental evaluation in Chapter 6.

The implementation of *one.world* currently runs on Windows and Linux PCs. It is largely written in Java, which provides us with a safe and portable execution platform. We use a small, native library to generate GUIDs, as they cannot be correctly generated in pure Java. Furthermore, we use the Berkeley DB [84] to implement reliable tuple storage. The Berkeley DB provides a trans-

actional hashtable on top of an unreliable file system and runs on a large number of operating systems, including Windows, VxWorks, and most Unix variants, thus ensuring the portability of our implementation. At the same time, to port *one.world* to smaller, handheld devices, such as HP's iPaq, we anticipate replacing the Berkeley DB with a more light-weight, in-memory database, because memory on these devices is already persistent. Our implementation currently lacks support for transactions as part of structured I/O (though, the individual structured I/O operations are fully implemented) and for loading code from environments. As a result, applications cannot use transactions to group several structured I/O operations into a single, atomic unit, and application code must be manually distributed across all devices that are expected to run an application. We expect that adding these features to our implementation will be straightforward. Our implementation does, however, include library support for building GUI-based applications, for a command line shell, which is accessible both locally and remotely (through the telnet protocol) and thus simplifies the management of the different devices in a pervasive computing environment, and for converting between files and stored tuples, thus enabling the exchange of data between conventional operating systems and *one.world*.

one.world has been released as an open source package and is currently at version 0.7.1. The implementation has approximately 19,000 non-commenting source statements (NCSS). Our entire source tree, including regression tests, benchmarks, and applications, has approximately 40,000 NCSS or 109,000 lines of well-documented code, representing an overall development effort of about six man years. A Java archive file with the binaries for *one.world* itself is 514 KB. The GUID generation library requires 28 KB on Windows and 14 KB on Linux systems, while the Berkeley DB libraries require another 500 KB on Windows and 791 KB on Linux systems.

Our implementation does not rely on features that are unique to Java. It requires a type-safe execution environment, support for reflection and object serialization, the ability to control how applications use threads, and the ability to customize the code loading process. As a result, *one.world* could also be implemented on other platforms that provide these features, such as Microsoft's common language runtime [104].

5.1 Environments as Application Containers

As described in the previous chapter, applications in *one.world* execute asynchronously and in isolation from each other. They can only interact with each other through asynchronous events and may only reference event handlers implemented by other environments, never arbitrary objects. The challenge in providing asynchrony and isolation is to implement these features on a platform, the Java virtual machine, that provides neither: Java is based on a synchronous execution model, favoring multi-threaded code, and does not isolate objects belonging to different applications.

While several projects have explored how to provide either asynchrony or isolation for Java, none provide both at the same time. Notably, SEDA [114] implements asynchrony by separating application code into so-called stages that have their own event queues and thread pools to process enqueued events. But SEDA does not provide isolation between stages (a non-goal for SEDA, which is targeted at building scalable Internet services). Furthermore, it exposes the event processing machinery to application developers, requiring that events be explicitly enqueued in the appropriate queues and thus making it unnecessarily hard to write asynchronous code. In contrast, KaffeOS [7] does provide isolation for applications running on the same virtual machine. However, while KaffeOS builds on nested class loaders [68] to isolate the code of different applications, it still requires extensive modifications to the underlying virtual machine and, as a result, is not portable. Furthermore, while KaffeOS does allow for objects to be shared between applications, its reliance on explicitly configured, shared heaps exposes a somewhat awkward programming model that more closely resembles shared memory in traditional operating systems than the capability-based sharing model of Java and other extensible systems such as SPIN [9]. Finally, by using proxy objects, which, comparable to RPC stubs [12], mediate access to objects in different protection domains, the J-Kernel [111] does provide isolation without requiring modifications to the Java virtual machine. However, because, by default, it relies on Java serialization to copy data between applications, it has a relatively high overhead.

Our solution to implementing asynchrony and isolation is based on the realization that, by leveraging our architecture's uniform event handling interface and tuple-based data model, we can combine elements from all three systems discussed above while also avoiding their limitations. More specifically, as in SEDA, our implementation of asynchrony is based on event queues and thread

pools. Similarly, as in KaffeOS, our implementation of isolation is based on nested class loaders. However, because of our architecture's uniform event handling interface, we do not need to force developers to explicitly enqueue events in the appropriate event queues or to explicitly create heaps for sharing data. Rather, as in the J-Kernel, we use dynamically created proxy objects [96] that, comparable to RPC stubs, mediate access to event handlers in different environments. Because these proxy objects are event handlers themselves, which reference the original event handlers, we call them *wrapped event handlers*. The wrapped event handlers automatically enqueue events into the appropriate queues and, because of our architecture's tuple-based data model, copy events between protection domains without falling back on Java serialization. By combining event queues and thread pools, class loaders, and event handler wrapping, our implementation cleanly layers asynchrony and isolation on top of a virtual machine that supports neither. However, just like SEDA, our implementation requires the ability to control how applications create and use threads, and, just like KaffeOS, our implementation requires the ability to customize the code loading process. We now describe the three implementation techniques in detail.

At the most basic, each environment processes events independently from other environments by using its own queue of pending *event handler, event* invocations and the corresponding thread pool to process the queue. Since (a)synchrony and concurrency are orthogonal concerns, the thread pool can either be fixed to a single thread (thus disallowing concurrency), or it can dynamically grow and shrink with load by using a thread pool controller, similar to the one used by SEDA. This way, an environment's thread pool adjusts to the actual load and consumes only as many threads (a limited resource for most operating systems) as actually necessary, up to a predetermined limit. Whether the thread pool is fixed to a single thread depends on the components instantiated in the environment. Each component declares whether it is concurrency-safe or not. If at least one component is *not* concurrency-safe, the environment's thread pool is fixed to a single thread; otherwise, it grows and shrinks with load.

one.world's thread pool controller resembles SEDA's controller as following. Each dynamically sized pool has a minimum and a maximum number of threads. To dynamically shrink a pool, the threads in the pool observe themselves. If a thread is idle for longer than a predetermined duration and there are more threads than the minimum number, the thread terminates itself. To dynamically grow a pool, we rely on a controller thread. The controller thread periodically scans all queues of

pending *(event handler, event)* invocations. If a queue is more than half full, the controller adds more threads to the corresponding thread pool. Unlike SEDA, whose event queues are of unlimited length, our architecture's queues of pending *(event handler, event)* invocations are of a fixed length (and we thus rely on a relative threshold instead of an absolute number of enqueued events to trigger the addition of threads). We believe that a fixed queue length better ensures a fair distribution of device resources between environments. In particular, a heavily overloaded environment cannot consume all available memory for its queue of pending *(event handler, event)* invocations.

In contrast to application environments, which all share the same thread pool configuration parameters, *one.world*'s kernel relies on a dynamically sized thread pool with a larger minimum and maximum number of threads. As the kernel provides services shared by all applications, we can reasonably expect it to encounter a larger load and to thus require additional resources for processing events. Furthermore, because TCP effectively establishes a synchronous link between two devices, structured I/O networking uses an additional, internal queue and thread pool (fixed to a single thread) for each TCP connection. This way, an unresponsive device or slow network connection cannot back up the entire kernel. Rather, the internal queue of pending *(event handler, event)* invocations for that TCP connection will fill, and applications sending tuples will be notified that the communications channel is overloaded.

Nested class loaders are used to separate the code of applications in different protection domains. They are arranged in a simple hierarchy, with one root class loader having a child class loader for every application's protection domain, similar to the class loader hierarchy in KaffeOS. The root class loader is responsible for loading core classes, including *one.world*'s kernel, and is created on startup. Child class loaders load applications' code, but defer to the root class loader for Java's platform classes and *one.world*'s core classes. Since child class loaders isolate code in different protection domains, their lifetimes are dependent on the applications' lifetimes. They are created when an application is loaded into an environment or when it migrates to a device, and they are destroyed when the application's environment is destroyed or when the application moves to another device. As a result, core classes are shared between protection domains and loaded once, and application code only needs to be in memory while an application is running.

Event handler wrapping ensures that events are automatically placed into the appropriate *(event handler, event)* queue and that arbitrary object references cannot be leaked between pro-

tection domains. The idea behind event handler wrapping is that an application cannot directly reference an event handler originating from another environment, but only a proxy object mediating access to the original event handler. It is comparable to the use of stubs in RPC systems or the use of proxy objects in the J-Kernel. An application initially accesses event handlers in other environments by linking with event handlers exported by components in other environments, such as its environment's request handler, which is implemented by the root environment. The linker performs the event handler wrapping as part of the linking process. Any additional wrapping is then performed by the wrapped event handlers themselves.

A wrapped event handler keeps internal references to the original, unwrapped event handler, the unwrapped event handler's environment—which we call the target environment because it receives events, and the environment using the wrapped handler—which we call the source environment because it sends events. When an event is sent to a wrapped event handler, the wrapped event handler first ensures that all event handlers referenced by the event are correctly wrapped. By using the tuple accessor methods shown in Figure 4.1 on page 31, it traverses all fields of the event, including the fields of nested tuples, and modifies event handlers in place as necessary. If the protection domains of the source and target environments differ, the wrapped event handler then copies the event. Events whose classes were loaded by the root class loader are simply copied, because their code is shared between all protection domains. Events whose classes were loaded by a child class loader are recreated using the class loader of the target environment's protection domain, because their code is not shared between protection domains. If the source environment is destroyed, the events' code is still accessible in the target environment's protection domain, thus preserving the isolation between the different protection domains. Because tuples are highly structured and tuple field types are limited, the event copy and recreation code can avoid using Java serialization; instead, it traverses the event and directly copies basic types, such as numbers and strings, and employs reflection to recreate arrays and nested tuples. After wrapping and, if necessary, copying the event, the wrapped event handler enqueues the unwrapped event handler and the resulting event in the target environment's *(event handler, event)* queue. If the target environment's *(event handler, event)* queue is full, it instead enqueues a failure notification in the source environment's *(event handler, event)* queue (hence the internal reference to the source environment).

Copying events and wrapping event handlers is sufficient to prevent protection domain leaks

because tuples and event handlers are the only entities applications can define themselves. All other types that can be used in tuples are defined by our architecture and cannot reference arbitrary objects. Note that wrapping event handlers in place before copying events does not represent a security hole. As described above, our implementation first wraps all event handlers referenced by an event in place, replacing the unwrapped versions with the corresponding wrapped ones, and only then copies the event. A malicious application, in a different execution thread, might thus change back an already wrapped event handler to an unwrapped one and consequently leak a direct event handler reference to a different protection domain. However, the event copy and recreation code only copies references to wrapped event handlers, but never unwrapped event handlers, thus avoiding this security hole.

Taken together, event queues and thread pools, nested class loaders, and event handler wrapping isolate environments from each other and provide a well-defined method for asynchronously communicating between them. By building on our architecture's uniform event handler interface and tuple-based data model, our implementation automates the copying and enqueueing of events and consequently exposes a clean and simple event-based programming interface to developers. Furthermore, it does not require modifications to the Java virtual machine and is completely portable, while also avoiding the relatively high overhead of Java serialization. In other words, our implementation provides both asynchrony and isolation on top of a virtual machine that provides neither, while also avoiding the major limitations of SEDA, KaffeOS, and the J-Kernel.

5.2 Reliable Event Exchanges through Operations

As described in Chapter 4.3, event delivery in *one.world* has only at-most-once semantics. In particular, the request/monitor mechanism may silently drop an event when both the source and target environment's event queues are full. Furthermore, as in other distributed systems, a remote receiver may fail before receiving the original request or before returning a response. Our reliance on asynchrony thus raises the question of how to *reliably* communicate through events. Traditionally, distributed systems have employed transactions [30, 71] to provide failure detection and recovery. However, because transactions also provide isolation and durability, they are too general and, as a result, too heavy-weight. As described in Chapter 4.3.2, the operation library provides a more

targeted and consequently more light-weight alternative for reliably communicating through events: It matches each request with exactly one response and automatically detects timeouts and performs retries. The challenge in implementing the operation library is to support all common event exchange patterns, including local and remote event exchanges as well as both sides for multi-round interactions.

Our solution is based on the realization that, because events are exchanged through a uniform event handler interface and because a request's closure is returned with the corresponding response, we can easily interpose on all interactions and rely on event closures for marking and tracking event exchanges. For regular event exchanges, we employ a technique we call *closure replacement*, under which the application-supplied closure for the request is replaced with the operation library's own closure and, before returning the corresponding response to the application, restored in the response. More specifically, the operation library replaces a request's closure with its own closure, simply a GUID, and stores the corresponding state, including the original closure, the starting time of the event exchange, and the number of retries left, in an internal table, keyed by that GUID. It also replaces the request's source event handler with its own source. When it receives a response on that event handler, it uses the response's closure to access the corresponding state in its table and restores the original closure before forwarding the response to the application. For remote events (which are used to perform *send* operations as shown in Figure 4.8 on page 45), the implementation also replaces the closure of the nested event (the *msg* event in the sample code). This is necessary because a response from the remote receiver will contain the nested closure while a failure notification created during event delivery will contain the closure of the remote event.

To detect timeouts, the operation library periodically scans its table for expired request/response interactions. For efficiency reasons, scans traverse the table entries along a doubly-linked list that is ordered by timeout. They are triggered by a single, periodic timer. While this results in a loss of timeout precision, it also is considerably more scalable than using a timer for each request/response interaction. If a request/response interaction has expired, the operation library either retries the request or, after all retries have been unsuccessful, notifies the appropriate application logic of the timeout.

Closure replacement by operations not only forms the basis for tracking event exchanges, it also provides applications with additional flexibility when compared to event exchanges that are

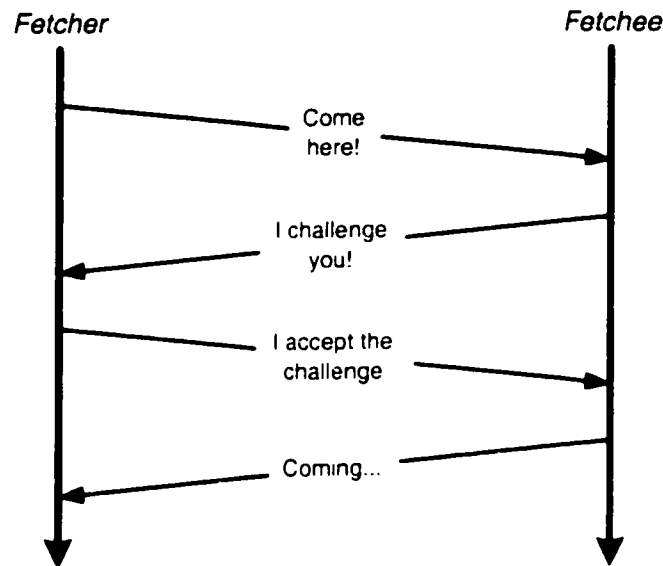


Figure 5.1: Illustration of Emcee’s fetcher protocol. The vertical arrows indicate the flow of time for the two protocol participants. The slanted arrows indicate the four messages exchanged by the fetcher and fetchee. This protocol is used by Emcee to fetch all of a user’s applications from a remote device (the fetchee) to the current device (the fetcher). At its core, it represents a simple challenge/response interchange.

not managed by operations. Without operations, an application may only use event closures that also have valid tuple field types. Furthermore, since closures are visible to the events’ receivers but typically hold state that is internal to an application and needed for processing responses, closures may leak sensitive application data. Closure replacement through operations avoids both limitations, because an application-supplied closure is immediately replaced by the operation’s own closure. As a result, the application-supplied closure is never tested for compliance with valid tuple field types—as compliance testing is only performed when copying events in a wrapped event handler—and never leaves the application’s protection domain. The implementation of Emcee, for example, relies on this feature of our operation library to use arbitrary objects as event closures, which simplifies Emcee’s event processing logic.

However, closure replacement breaks down when operations are used on both sides of multi-round interactions because the operations are not nested within each other. To illustrate the problem

with closure replacement for multi-round interactions, consider the fetcher protocol illustrated in Figure 5.1, which is used by Emcee to fetch all of a user's applications. After the fetcher announces its intent to fetch a user's applications through the "Come here!" message, the protocol authenticates the user through a simple challenge/response interaction. The fetchee issues a challenge in the "I challenge you!" message, and the fetcher returns a new value computed from the challenge and a shared secret (the user's password) in the "I accept the challenge" message, thus authenticating the user. The "Coming..." message concludes the protocol by confirming a valid response. The fetcher's implementation of this protocol uses an operation to send the "Come here!" and the "I accept the challenge" messages, and the fetchee's implementation uses an operation to send the "I challenge you!" message. However, even if the fetchee correctly copies the closure from the "Come here!" message into the "I challenge you!" message, its use of the operation library results in the closure being replaced and the fetcher's operation never seeing its own closure. As a result, the protocol times out and never completes.

To address this problem, the operation library supports a special type of closure, called a *chaining closure*. A chaining closure is a tuple with no additional fields besides the ID and metadata fields common to all tuples. The contract is that when the original closure of a request is a chaining closure (or a subtype), the operation library does not replace the closure. Rather, it passes the provided closure through and tracks the request/response interaction based on the ID of the chaining closure. As a result, applications can implement multi-round interactions while still relying on operations to detect timeouts and perform retries on both sides. In the above example, the fetcher creates a new chaining closure for the "Come here!" message, which is then passed from message to message (hence the name) and never replaced.

By using closures to mark and track event exchanges, our implementation of the operation library reliably manages request/response interactions. It matches each request with exactly one response and automatically detects timeouts and performs retries, thus simplifying the development of event-based applications. However, because closures are visible to the application developer, the use of the operation library also requires some developer discipline. Notably, depending on the type of event exchange managed by the operation library, a developer may use any closure—for regular event exchanges—or must use chaining closures—for multi-round interactions. Furthermore, when implementing service functionality, a developer must ensure that the request's closure is copied into

the response; though, *one.world*'s library support for implementing event handlers does this automatically. Overall, in our experience with using the operation library, the advantages of having reliable event exchanges far outweigh the discipline required in using the library.

5.3 *Discovery and Remote Event Passing*

As described in Chapter 4.4, discovery and remote event passing provide the ability to locate resources and send events to them. The single most important requirement for implementing the two services is that discovery be almost always available. After all, if discovery is not available, applications cannot locate necessary resources and thus cannot adapt to changes in their runtime context. Our implementation relies on a centralized server, which holds the discovery directory and, to ensure availability, is automatically elected from all devices running *one.world* on the local network. Discovery server elections eliminate the need for manual configuration and administration and thus make it possible to use discovery outside well-managed computing environments, such as a conference site. With discovery server elections, the conference attendees can exchange contacts and collaborate on presentations, simply by enabling their devices' wireless capabilities.

The challenge in implementing discovery server elections is that they must be called early and complete quickly, so that discovery adjusts swiftly to changes in device and network topology. As a result, we cannot rely on traditional consensus algorithms. Notably, the Paxos algorithm, an optimal algorithm for reaching consensus in asynchronous networks [61, 62], requires answers from a majority of participants and may thus introduce considerable delays. Furthermore, Paxos already requires an elected leader, exactly what we are trying to implement. Our solution is based on the realization that pervasive applications do not require consensus on which discovery server to use; they only require that the server actually used has the complete discovery directory. In other words, we can implement elections so that they are called early and complete quickly, even though this may lead to inconsistencies, as long as we also make discovery tolerate such inconsistencies.

However, before discussing our implementation of discovery in detail, we first review the implementation of remote event passing, as discovery builds on REP. To provide point-to-point communications, REP relies on a table mapping the GUIDs and names encapsulated by symbolic handlers to actual event handlers. Mappings are added through the *export* operation and removed when the

corresponding leases are canceled or expire. To send an event through point-to-point communications, REP forwards the event to the device specified by the symbolic handler, where the GUID or name is resolved to an event handler by performing a table lookup, and the event is delivered to that event handler. The implementation uses structured I/O networking to communicate events between devices, across either UDP or TCP. REP defaults to TCP, but senders can override this default when sending an event. The choice of UDP or TCP affects the reliability and timeliness of event delivery between devices, but has no other application-visible effect. For TCP-based communications, REP maintains a cache of connections to avoid recreating connections. Furthermore, sending events across the network is avoided altogether if the sender and the receiver are on the same device [8].

Discovery is implemented on top of REP's point-to-point communications and is split into three components: a discovery client, which services all applications running on a device, a discovery server, which holds the discovery directory, and an election manager, which ensures that the discovery server is available. Consistent with their roles, the discovery client and election manager run on every device, while the server usually runs on only one device for a local network. In summary, elections are called aggressively and terminate after a fixed time period, which ensures that a discovery server is almost always available but can also lead to more than one elected discovery server (though, never none). To tolerate such inconsistencies, the discovery directory held by the server is treated as soft state. Only clients hold the authoritative state, which they forward to *all* visible servers. However, when servicing discovery requests, they only consult *one* visible server.

In detail, the discovery client is responsible for maintaining the *(event handler, descriptor)* bindings for all applications running on a device and thus holds the authoritative version of that state. Additionally, it is responsible for forwarding discovery requests to the discovery server. To maintain discovery bindings, the discovery client uses an internal table. Just as for point-to-point communications, bindings are added through the *export* operation and removed when the corresponding leases are canceled or expire. When a discovery server becomes visible on the local network, the discovery client propagates all bindings to the server, thus ensuring that every server on the local network sees the device's bindings. Server-side bindings are leased and the discovery client maintains these internal leases. To forward discovery requests, the discovery client simply sends the requests to one of the currently visible servers.

The discovery server is responsible for actually servicing discovery requests. It accepts the

(*event handler, descriptor*) bindings propagated by discovery clients and integrates them into a single table for all applications on the local network, which, in contrast to the clients' tables, is soft state. Descriptors that are identical, ignoring tuple IDs and metadata, are collapsed into a single table entry pointing to multiple remote references to improve the performance of query processing. When processing a *resolve* operation, the discovery server looks up matching services and returns the result to the discovery client that forwarded the operation. When processing a late binding *send* operation (which, as described in Chapter 4.4, combines a discovery lookup with the routing of an event into a single operation), the discovery server first processes observing reverse lookups on the event and forwards the event to all matching services. It then processes forward lookups on the resource descriptors as well as consuming reverse lookups on the event and forwards the event to one matching service for anycast and to all matching services for multicast. For both operations, if no service matches, a failure notification is reflected back to the sending application.

The election manager is responsible for ensuring that a discovery server is present on the local network. The current discovery server periodically announces its presence, every second in our implementation. Announcements are sent as UDP multicasts through structured I/O networking. The election manager listens for discovery server announcements. If it does not receive announcements for two announcement periods, it calls an election, thus allowing for one but not more consecutively lost announcements. During an election, each device broadcasts a score representing the device's suitability for hosting the discovery server. In our implementation, this score is computed from a device's uptime and memory size, as both statistics are readily available within Java and, taken together, favor larger and more stable devices. Obviously, this heuristic should be improved by also accounting for CPU speed and available network bandwidth, even though they are not as readily available within Java. During the election, each device also observes the other devices' broadcasts and keeps track of the currently highest score. The election is terminated after a fixed period, one second in our implementation, and the device with the highest score starts the discovery server.

Since each device already tracks the currently highest score, our implementation includes an optimization, under which a device only broadcasts its score if it is higher than the currently highest score. Consequently, the cost of electing a discovery server on a network with n devices is at most $n + 1$ broadcast messages, one message to start the election and at most one message per device to announce its score. This cost is small, as long as the cost of sending a broadcast message is com-

parable to the cost of sending a unicast message—which is the case for most local area networking technologies, including Ethernet for wired networks and 802.11 [38] for wireless networks. However, the cost of sending broadcast messages also limits the number of devices that can participate in an election, as at most $n + 1$ messages need to fit into an election period. If more devices try to participate in an election, a new discovery server will still be elected; though, it may not be the device with the highest score, as that device may not have an opportunity to broadcast its score.

To reduce the period during which discovery may be unavailable, our implementation includes additional optimizations, under which the discovery client and server proactively call elections without waiting for two missed server announcements. In particular, the discovery client calls an election when it receives a malformed or unexpected event indicating that the current discovery has failed. Furthermore, the current discovery server calls an election when its device is about to be shut down.

Overall, the discovery service can be in one of three states, depending on how many discovery servers are currently running within the network:

No discovery server. This is an exceptional state, under which discovery is temporarily unavailable. This state can be reached when the device running the server crashes or when the network becomes partitioned. After two missed announcements, the election manager on one of the remaining devices initiates an election and a new discovery server is started. So, after a maximum period of three seconds—two missed announcements, each one second apart, and an election one second long, at least one discovery server is running again on the network. Clients see the announcements from the new discovery server and forward their bindings to the server. No discovery state is lost, because the discovery server’s table is only soft state.

One discovery server. This is the normal state, and discovery works as expected. The current server adds new *(event handler, descriptor)* bindings to its table, removes expired bindings, and processes requests as they are forwarded by clients. A device entering the local network sees the current server’s announcement within one second, forwards all current bindings to the server, and thus becomes part of the local network.

More than one discovery servers. This is an exceptional state, though discovery is fully available. This state can be reached when two network partitions are merged, or when messages are

lost during an election and two devices start a discovery server. It is also entered when the discovery server proactively calls an election as its device is shut down. Discovery remains fully available, because clients export their *(event handler, descriptor)* bindings to all visible servers but forward requests to only one server. When a discovery server sees an announcement from another server with a higher score (which usually happens within one second), the server with the lower score shuts itself down and the network returns to the normal state.

Our server-based implementation of discovery is largely transparent to applications. Only a server crash or a network partition result in a short transitional period, during which discovery is effectively unavailable as discovery lookups do not return any results—after all, there is no discovery directory. This transitional period could usually be hidden by using more than one discovery server in the normal case. Note that modifying the implementation to utilize more than one server in the normal case represents a relatively simple change, because discovery clients already work with more than one server. Further note that using more than one server can also improve the scalability of our discovery service by load balancing requests across several servers.

By relying on an elected discovery server, our implementation of *one.world*'s discovery service ensures that discovery is almost always available. Elections are called aggressively and, in contrast to consensus algorithms which require answers from a majority of participants, complete after a fixed period. Elections may thus result in a less than optimal selection—when the device with the highest score did not have an opportunity to broadcast its score—or even to more than one elected server—when broadcasts were not seen by all devices. To tolerate such inconsistencies, a server's state is treated as soft state, and clients, which hold the authoritative version, forward their state to all visible servers. Though, they forward requests to only one visible server. Our implementation is limited by the cost of sending broadcast messages, as, on a network with n devices, at most $n + 1$ messages need to fit into an election period.

5.4 Checkpointing and Migration

one.world's checkpointing and migration services are novel in that they leverage our architecture's environment hierarchy to limit the state that is checkpointed and migrated, respectively: an application's components and data are preserved while references to resources outside the tree are not.

This design provides application developers with a simple and well-defined model for what state is affected by the two services. It also marks a new and, we believe, reasonable trade-off between service features and implementation complexity. In particular, visibly erasing outside references not only avoids much of the complexity of transparent migration systems, such as Sprite [28] or Emerald [56, 98], but also enables our services to operate under weak or intermittent connectivity—when migrating from one device to another, the two devices only need to be connected while the move or copy operation is in progress. Furthermore, besides erasing outside references, our migration service migrates an application’s entire state, including its execution state and persistent data, thus providing more functionality than most agent systems, such as IBM’s Aglets [63], which only migrate an application’s objects but not its execution state nor persistent data. We provide a detailed comparison with other migration systems in Chapter 8. In this section, we describe how our architecture captures and restores the in-memory state of an environment tree and then present how the checkpointing and migration services build on this shared functionality.

At the core of checkpointing and migration lies the ability to capture and restore the in-memory state of an environment tree. The challenge in implementing this shared functionality is to preserve the state of the entire tree, even if the tree spans several protection domains, while also erasing references to outside resources. By comparison, previous migration systems either move one process, and therefore one protection domain, at a time—as is the case for Sprite—or do not provide isolation at all—as is the case for Emerald and Aglets. Furthermore, as noted above, Aglets does not even migrate an application’s execution state. Our implementation meets this challenge through two techniques: it inspects wrapped event handlers to determine which references to null out and annotates Java classes with their class loader to preserve protection domains. Our implementation directly builds on the implementation of environments as application containers (as described in Chapter 5.1) and, consequently, does not require any additional mechanism. In other words, by implementing protection on top of the virtual machine, capturing the state of several protection domains in one consistent checkpoint becomes manageable. Similarly, by implementing asynchrony through explicit event queues and thread pools, capturing the applications’ execution state becomes feasible.

When capturing the in-memory state of an environment tree, our implementation first quiesces all environments in the tree, so that it can capture a consistent snapshot. In particular, it lets all cur-

rently active event handler invocations run to completion and prevents new invocations from being executed. Once the environment tree is quiesced, our implementation serializes each environment's main and monitor handlers as well as *(event handler, event)* queue. During serialization, our implementation inspects every wrapped handler and nulls out those handlers whose target environment is not in the environment tree, thus preserving references between environments in the tree but not to environments outside the tree. Note that our implementation does serialize request handlers belonging to environments in the tree (even though their target environment is the root environment), so that applications can still communicate with the kernel and the outside world after restoration of the checkpoint. Furthermore, Java classes are annotated with their protection domain, that is, class loader. When deserializing a Java class, the class can then be loaded by the appropriate class loader, thus preserving the protection domains in the environment tree. Once serialization is complete, the environment tree is reactivated. The resulting checkpoint is represented as a *CheckPoint* tuple, which contains the serialized application state, the identifiers for the environments in the checkpoint, and a timestamp. When restoring a checkpoint, our implementation simply deserializes the application state contained in the *CheckPoint* tuple, enqueues an appropriate notification at the beginning of each environment's *(event handler, event)* queue, and then reactivates the environments in the tree.

The implementation of checkpointing uses structured I/O to read and write *CheckPoint* tuples. After creating a checkpoint, it writes the checkpoint to the root of the checkpointed environment tree by using a structured I/O *put* operation. The checkpoint thus becomes a part of the application and, for example, is moved with the application when the application is migrated. When restoring a checkpoint, our implementation can either restore a checkpoint with a specific timestamp or the latest checkpoint. A specific checkpoint is read using a structured I/O *read* operation that queries for a *CheckPoint* tuple with the specified timestamp. The latest checkpoint is read by using a structured I/O *query* operation and then iterating over all *CheckPoint* tuples to determine the latest checkpoint.

The implementation of migration relies on a network protocol to communicate the migrating environment tree, including the environments' metadata, stored tuples, and in-memory state (as represented by the previously generated *CheckPoint* tuple), from one device to another. The protocol is implemented using our architecture's remote events and is organized into several rounds, where

each event issued by the sender is confirmed by the receiver, thus providing a very simple form of flow control. Both sender and receiver use operations to manage the event interchanges, employing a `ChainingClosure` as described in Chapter 5.2. The sender's operation connects each request to its response, while the receiver's operation connects each response to the next request. If an event is lost or an error occurs, the protocol and consequently the migration are aborted. The protocol is described in detail in Appendix A.

Our implementation does not yet migrate program binaries, that is, the applications' Java class data. Rather, Java classes are loaded from a device's local classpath and not migrated. In contrast, *one.world*'s design calls for storing class data as tuples in an application's environment and then migrating the class data with the application. However, we believe that this design may be too simplistic, as class data shared between applications is stored several times and may be repeatedly migrated. We thus suggest an improved design that still stores class data as tuples in individual environments but also backs class data by a shared cache, effectively storing each class only once. With the improved design, class data is only migrated if the necessary classes are not already in the cache.

The complexity of *one.world*'s migration service, as measured by its code size, is quite reasonable. With approximately 1,600 statements, it amounts to only 8% of the overall kernel sources. In particular, the code for checkpointing and the migration protocol comprises about 1,200 statements out of about 2,500 statements for all environment operations. The code for migrating stored tuples comprises an additional 300 statements out of about 2,000 statements for all tuple storage operations. Furthermore, we incur some cost for making all core objects serializable, typically less than 10 relatively formulaic statements per class. At the same time, *one.world*'s support for migration, just as for previous migration systems, does interact with several aspects of our architecture's kernel, resulting in five tightly interdependent classes at the core of *one.world*. Note that application classes also need to be serializable; though, making them serializable is typically easier than for *one.world*'s core objects, requiring no or very little additional code in the common case.

By building on our implementation of asynchrony and isolation, the implementation of *one.world*'s checkpointing and migration services can directly capture an environment tree's state while also erasing references to outside resources. As a result, and in contrast to previous migration systems, our implementation can easily checkpoint and migrate several protection domains in one

Table 5.1: Summary of implementation challenges. *Feature* specifies the property of our architecture. *Challenge* describes why that feature is hard to implement, and *Insight* describes the basic realization behind our solution. *Solution* lists the specific techniques used to meet the challenge.

Feature	Challenge	Insight	Solution
Asynchrony & isolation.	Java virtual machine provides neither.	We can easily interpose on all event-based interactions and thus control how applications interact.	Utilize per-environment event queues and thread pools to implement basic asynchrony; load code with distinct class loaders to separate applications in different protection domains; wrap event handlers passed between environments to automate enqueueing of events and to enforce isolation.
Reliable event delivery.	Basic event delivery has at-most-once semantics.	We can mark and track all event exchanges.	Interpose on client's interactions with outside services; replace event closures to track event exchanges, providing automatic timeouts and retries; use <code>ChainingClosure</code> for both sides of multi-round interactions.
Availability of discovery.	Traditional consensus algorithms are too heavyweight and may already require a leader.	We do not require consistency, only fast elections.	Rely on elected discovery server that holds only soft state; call elections early and finish them after a fixed period; make clients tolerate inconsistencies by exporting bindings to all servers but by forwarding requests to only one server.
Completeness of checkpoints.	Checkpoints must preserve internal structure of environment tree but cannot retain outside references.	We already have access to all necessary information.	Inspect wrapped event handlers and erase only event handlers implemented by environments outside the tree; annotate classes with their protection domains and load them with the corresponding class loader while restoring the checkpoint.

operation. Furthermore, in contrast to many mobile agent systems, it can easily capture the environment tree's execution state. However, because execution stacks are implicitly provided by the underlying virtual machine, it can only capture the execution state of pending *(event handler, event)* invocations, but not currently executing ones. Capturing the state of currently executing *(event handler, event)* invocations would require modifications to the virtual machine.

5.5 Summary

In this chapter, we provided an overview of *one.world*'s implementation and then discussed its more interesting aspects in detail. As summarized in Table 5.1, each of the previous four sections presented a particular implementation challenge and the corresponding solution. In particular, to provide asynchrony and isolation on top of the Java virtual machine, our implementation uses event queues and thread pools, nested class loaders, and event handler wrapping. To provide reliable event delivery on top of unreliable events, our implementation interposes on clients' interactions with outside services and relies on closure replacement and chaining closures to track these interactions. To ensure the availability of discovery without running a traditional consensus algorithm, our implementation relies on an elected discovery server that holds only soft state. Elections are called early and complete within a fixed time period. Discovery clients tolerate any resulting inconsistencies by exporting bindings to all servers while forwarding requests to only one server. Finally, to create a checkpoint that preserves the structure of an environment tree while also erasing outside references, our implementation inspects wrapped event handlers, only nulling out those implemented by environments outside the tree, and annotates classes with their protection domains, thus making it possible to load them with the appropriate class loader while restoring the checkpoint.

Chapter 6

EXPERIMENTAL EVALUATION

In this chapter, we introduce the user-space services, utilities, and applications we and others have built and present the results of our experimental evaluation, which is based on these programs. The goal is to answer the question of whether *one.world* is good enough for building pervasive applications. Or, to be more in line with the hypothesis presented in Chapter 1, we are trying to determine whether focusing on the unique requirements of pervasive computing has resulted in a system architecture that enables developers to effectively build adaptable applications so that users do not have to constantly reconfigure their systems. However, both questions, by themselves, are rather general and hard to answer. Accordingly, we rely on four, more specific criteria and corresponding questions to evaluate our architecture:

Completeness. Can we build useful programs using *one.world*'s primitives? This criterion determines whether our architecture is sufficiently powerful and extensible to support interesting user-space programs, including additional services and utilities akin to the Unix shell.

Complexity. How hard is it to write code in *one.world*? This criterion determines the effort involved in developing programs for our architecture. We are especially interested in how making applications adaptable impacts programmer productivity.

Performance. Is system performance acceptable? This criterion determines whether our architecture performs well enough to support actual application workloads. Since our goal is to make applications adaptable, we are especially interested in whether applications respond quickly to changes in their runtime context.

Utility. Have we enabled others to be successful? This criterion determines whether others can build real pervasive applications on top of *one.world*. It also represents the most important

criterion. After all, a system architecture is only as useful as the programs running on top of it.

We address these four criteria in the rest of this chapter, one criterion per section, and introduce the user-space programs we and others built for our architecture along the way, presenting our own programs in Chapter 6.1 and the programs others built in Chapter 6.4. To summarize the results, we show that *one.world* (1) is sufficiently complete to support interesting programs on top of it, (2) is not significantly harder to program than with conventional programming styles, (3) has acceptable performance, with applications reacting quickly to changes in their runtime context, and, most importantly, (4) enables others to successfully build pervasive applications. In other words, our experimental results show that *one.world* does, in fact, enable developers to build applications that adapt to change instead of forcing users to constantly reconfigure their systems and, consequently, they confirm the hypothesis behind this dissertation.

6.1 Completeness

To evaluate our architecture, including to determine completeness, we built a set of user-space programs. In this section, we first describe their functionality and implementations and highlight how they utilize *one.world*'s services. In particular, we present a replication service in Chapter 6.1.1, followed by Emcee—our user and application management utility—as well as Chat—our text and audio messaging application—in Chapter 6.1.2. We then discuss the results regarding completeness in Chapter 6.1.3.

6.1.1 Replication Service

To provide ubiquitous access to people's information, pervasive applications need to access the corresponding data items, even if several people share the same data and access it from different and possibly disconnected devices. One important strategy for providing this capability is to replicate the data. Our replication service does just that and makes stored tuples accessible on multiple devices that may be disconnected. By providing replication as a common application building block, our replication service simplifies the development of pervasive applications, as developers need not reimplement this important, but also complex capability.

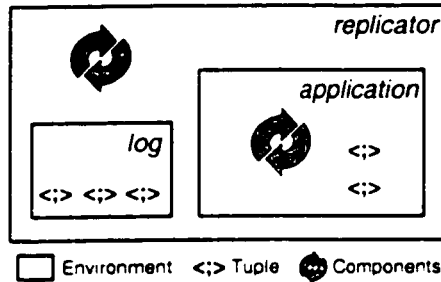


Figure 6.1: Illustration of our replication service’s structure. The *replicator* environment intercepts all storage operations issued by the application. In disconnected mode, the replicator logs updates in the *log* environment. In connected mode, it directly forwards them to the master.

Our replication service is patterned after Gray et al.’s two-tier replication model [41]. A master node owns all data and replicas have copies of that data. Replicas can either be connected or disconnected. In connected mode, updates are final and performed directly on the master. In disconnected mode, updates are tentative and logged on the replica. When a replica becomes connected again, it synchronizes with the master by replaying its log against the master and by receiving updates from the master. The replica may then disconnect again or continue in connected mode.

We chose two-tier replication over Bayou’s epidemic replication model [88, 102] for two reasons. First, two-tier replication is easier to explain to users, as tentative updates may only change once, during synchronization, and not repeatedly. Relying on an easy-to-explain model for replication is important, because pervasive computing is expressly targeted at supporting all people and not just computer experts. Second, on a more technical level, two-tier replication avoids system delusion [41]. Delusion occurs when large numbers of replicas reconcile with each other repeatedly in the absence of a master and consequently diverge further and further from each other.

The implementation of our replication service runs in user-space and, as illustrated in Figure 6.1, exploits the environment nesting—through the request/monitor mechanism—to interpose on an application’s access to tuple storage. The replicator logs updates in the *log* environment when in disconnected mode and forwards them to the master when in connected mode. On reconnection of a disconnected device, instead of sending individual updates as remote events, the log is sent to the master in one operation by copying the log environment. Similarly, updates are sent from the master

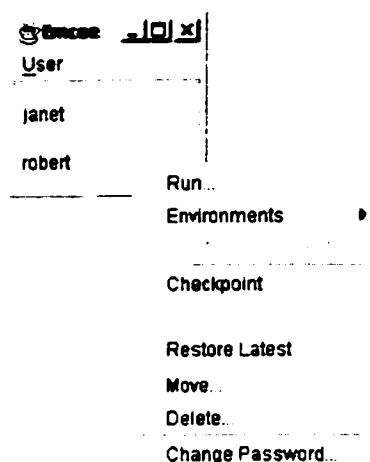


Figure 6.2: A screenshot of Emcee’s user interface. The main window lists the users whose applications run on the device. A popup menu for each user, shown for the user named *robert*, is used to perform most operations, such as running a new application or checkpointing a user’s applications. The user menu supports the creation of new users and the fetching of a user’s applications from another device.

to the replica by migrating an environment containing such updates.

As illustrated by our replication service, migration can serve as an internal building block for applications and can be used to simplify communications. Furthermore, because environments host both computations and data, migration provides an effective way to move application-specific reconciliation logic to the master: the replicator simply instantiates the necessary components in the log environment before copying it. Finally, our replication service is not limited to using migration internally; rather, the master and its replicas are migratable themselves. Migrating the master is useful when, for example, upgrading the computer the master is running on; migrating a replica is useful when the user is switching devices.

6.1.2 *Emcee and Chat*

Emcee, whose user interface is shown in Figure 6.2, manages users and their applications. It includes support for creating new users, running applications for a user, and checkpointing all of a user’s applications. Emcee also provides the ability to move or copy applications between users, simply by dragging an application’s flag icon, as shown in the upper right corner of Figure 6.3 on page 76.

and dropping it onto a user's name in the main window. Finally, it supports moving all of a user's applications between devices and thus helps realize the vision of a computing environment, in which applications follow a user as she moves through the physical world. Applications can either be pushed from the current device to another device, or they can be pulled from another device to the current device. Emcee can manage any *one.world* application: an application does not need to implement any features specific to Emcee. However, to support drag and drop through the flag icon, an application's developer needs to add three lines of code to the application.

The implementation of Emcee structures the environment hierarchy according to the pattern `/User/<user>/<application>`. Emcee runs in the `/User` environment and uses a child environment for each user and a grandchild for each application. Each user's root environment stores that user's preferences, including her password, and application checkpoints. The implementation of most operations is straight-forward, since they directly utilize *one.world*'s primitives (as illustrated in Figure 4.9 on page 50). The exception is fetching a user's applications from a remote device. As already discussed in Chapter 5.2, it uses a two-round protocol to authenticate the user to the remote instance of Emcee that is currently hosting the user's applications. After the user has been successfully authenticated, the remote Emcee initiates a migration of the user's environment tree to the requesting device. If the user's location is not specified, the initial remote event for the fetcher protocol is routed through late binding discovery. Otherwise, it is sent directly to the remote device (see Figure 4.8 on page 45).

Chat, whose user interface is shown in Figure 6.3, provides text and audio messaging. It is based on a simple model, under which users send text and audio messages to a channel and subscribe to a channel to see and hear the messages sent to it. The implementation sends all messages through late binding discovery, using TCP-based communications for text messages and UDP-based communications for audio messages. For each subscribed channel, Chat exports an event handler to discovery, which then receives the corresponding messages. Audio can either be streamed from a microphone or from sound tuples stored in an environment. Since music files tend to be large, they are converted into a sequence of audio tuples when they are imported into *one.world*. Using the tuple IDs as symbolic references, the sequence of audio tuples forms a doubly-linked list. As Chat is streaming audio messages, it traverses this list and reads individual tuples on demand, buffering one second of audio data in memory.

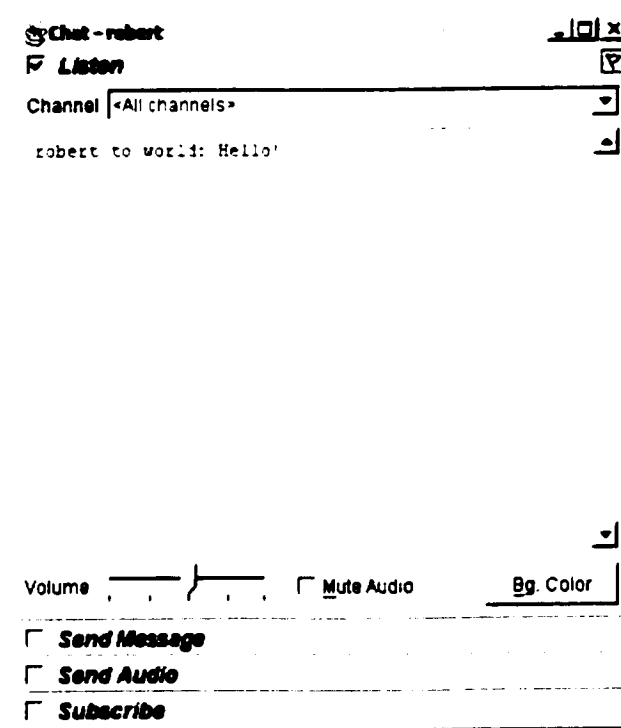


Figure 6.3: A screenshot of Chat's user interface. The user interface is divided into four panels, which can be independently expanded or collapsed by checking or unchecking the corresponding checkbox. The *listen* panel shows received text messages and provides volume controls for audio playback. The *send message* panel lets the user send text messages and the *send audio* panel lets the user send audio, either from a microphone or from stored audio tuples. Finally, the *subscribe* panel lets the user select the channels he or she is currently listening to.

Emcee and Chat illustrate the power of migration combined with dynamic composition through discovery and environment nesting. Discovery connects applications in the face of migration. Because Chat uses late binding discovery to route text and audio messages, messages are correctly delivered to all subscribed users even if the users move through the physical world. At the same time, environment nesting makes it possible to easily migrate applications, such as Chat, that have no migration logic of their own. Emcee controls the location of a user's applications simply by nesting the applications in its environment. Chat does not need its own migration logic and can automatically benefit from future improvements in Emcee's migration support, such as using smart badges to identify a user's location instead of requiring the user to explicitly move and fetch appli-

cations.

6.1.3 Discussion

The above programs (and the applications presented later on in this chapter) clearly show that our architecture is powerful enough to support a variety of useful programs. Furthermore, our replication service and Emcee, both of which directly help in realizing the vision of ubiquitous access to a person's data and applications, demonstrate that services and utilities can indeed be implemented in user-space. The key feature for enabling user-space services and utilities is *one.world*'s environment hierarchy. Nested environments make it easy to control other programs and, through the request/monitor mechanism, to interpose on their request streams. However, as discussed in Chapter 4.2.2, we had to carefully design the interface for structured I/O to support layering, an effort akin to previous work on stackable file systems [48].

Furthermore, in the course of developing the above programs, we did encounter one relatively minor limitation to *one.world*'s APIs: the performance evaluation of our replication service [44] suggests that the durability guarantees of structured I/O storage can result in too high a overhead for some applications. In particular, immediately forcing each *put* operation to disk is unnecessary when logging updates in disconnected mode, because all updates are already tentative. We thus designed (but not yet implemented) a simple extension to structured I/O, under which applications can optionally request that the destructive *put* and *delete* operations provide only relaxed durability guarantees and are lazily written to disk. Just as with traditional file system interfaces, applications using this option need to explicitly perform a *flush* operation to force pending updates to disk.

6.2 Complexity

To evaluate the effort involved in writing adaptable applications, we analyzed the process of implementing Emcee and Chat. The general theme for developing Emcee and Chat was that "no application is an island". Consistent with a computing environment where people and devices keep coming and going, applications need to assume that their runtime context changes quite frequently and that external resources are not static. Furthermore, they need to assume that their runtime context may be changed by other applications. These assumptions have a subtle but noticeable effect on the

implementations of Emcee and Chat. Rather than asserting complete control over the environments nested in the */User* environment, Emcee dynamically scans its children every second and updates the list of users in its main window accordingly. Similarly, it scans a user's environments before displaying the corresponding popup menu (which is displayed by selecting the "Environments" menu entry shown in Figure 6.2).

For Chat, these assumptions show up throughout the implementation, with Chat verifying that its internal configuration state is consistent with its runtime context. Most importantly, Chat verifies that the user, that is, the parent environment's name, is still the same after activation, restoration from a checkpoint, and migration. If the user has changed, it updates the user name displayed in its title bar, adjusts default channel subscriptions, and clears its history of text messages. Furthermore, it runs without audio if it cannot initialize the audio subsystem, but retains the corresponding configuration state so that it can resume playback when migrating to a different device. It also silences a channel if the audio tuples have been deleted from their environment. Finally, before processing any event, including text and audio messages, it checks for concurrent termination.

In our experience with Chat and Emcee, programming for change has certainly been tractable. The implementation aspects presented above are important for Emcee's and Chat's correct operation, but are not overly complex. Furthermore, programming for change can also simplify an application's implementation. For example, when Emcee fetches a user's applications, it needs some way to detect that the user's applications have arrived on the local device. But, because Emcee already scans its children every second, the arrival will be automatically detected during a scan and no additional mechanism is necessary. To put it differently, the initial effort in implementing an adaptable mechanism—dynamically scanning environments—has more than paid off by simplifying the implementation of an additional application feature—fetching a user's applications.

To quantify the effort involved in building Emcee and Chat, we tracked the time spent developing the two programs. They were implemented by three developers over a three month period. During that time, we also added new features to *one.world*'s implementation and debugged and profiled the architecture. Overall, implementing Emcee and Chat took 256 hours; a breakdown of this overall time is shown in Table 6.1. *Learning Java APIs* is the time spent for learning how to use Java platform APIs, notably the JavaSound API utilized by Chat. *User interface* is the time spent for implementing Emcee's and Chat's GUI. *Logic* is the time spent for implementing the actual

Table 6.1: Breakdown of development times in hours for Emcee and Chat. The times shown are the result of three developers implementing the two applications over a three month period. The activities are discussed in the text.

Activity	Time
Learning Java APIs	21.0
User interface	47.5
Logic	123.5
Refactoring	6.0
Debugging and profiling	58.0
Total time	256.0

application functionality. *Refactoring* is the time spent for transitioning both applications to newly added *one.world* support for building GUI-based applications. It does not include the time spent for implementing that support in our architecture, as that code is reusable (and has been reused) by other applications. Finally, *debugging and profiling* is the time spent for finding and fixing bugs in the two applications and for tuning their performance.

Since Emcee and Chat have 4,231 non-commenting source statements (NCSS), our overall productivity is 16.5 NCSS/hour.¹ As discussed above, *one.world* is effective at making programming for change tractable. In fact, adding audio messaging, not reacting to changes in the runtime context, represented the biggest challenge during the implementation effort, in part because we first had to learn how to use the JavaSound API. We spent 125 hours for approximately 1750 NCSS, resulting in an approximate productivity of 14 NCSS/hour. If we subtract the time spent learning Java platform APIs (including the JavaSound API), working around bugs in the Java platform, and refactoring our implementation from the total time, our overall productivity increases to 20.4 NCSS/hour, which represents an optimistic estimate of future productivity. Our actual productivity of 16.5 NCSS/hour lies at the lower end of the results reported for considerably smaller and simpler projects [92], but is almost twice as large as the long-term results reported for a commercial company [32]. Based on this, we extrapolate that programming for change does not decrease overall programmer productivity and conclude that it is not significantly harder than more contentional programming styles.

¹Productivity is traditionally measured in lines of code per hour or LOC/hour. NCSS/hour differs from LOC/hour in that it is more exact and ignores, for example, a brace appearing on a line by itself. As a result, NCSS/hour can be treated as a conservative approximation for LOC/hour

6.3 Performance

To determine whether our implementation performs well enough for real application usage, we measured the scalability of migration and late binding discovery. Migration and discovery are the two services the programs discussed in this chapter rely on the most and, in general, are indispensable for realizing applications that follow people through the physical world. Furthermore, to characterize system and application reactivity, we explored how Chat reacts to changes in its runtime context. Reactivity is especially important for pervasive applications, as they need to continuously adapt to a highly dynamic computing environment. It also marks a clear point of departure from traditional distributed applications such as Microsoft's Outlook, which locks up for minutes when it cannot reach the corresponding Exchange server.

All measurements reported on in this chapter were performed using Dell Dimension 4100 PCs, with Pentium III 800 MHz processors, 256 MB of RAM, and 45 or 60 GB 7,200 RPM Ultra ATA/100 disks. The PCs are connected by a 100 Mb switched Ethernet. We use Sun's HotSpot client virtual machine 1.3.1 running under Windows 2000 and Sleepycat's Berkeley DB 3.2.9.

To quantify the scalability of migration, we conducted a set of micro-benchmarks. For the micro-benchmarks, we use a small application that moves itself across a set of devices in a tight loop. We measure the application circling 25 times around three PCs for each experiment. To test the scalability of migration under different loads, we add an increasing number of tuples carrying 100 bytes of data, tuples carrying 100,000 bytes of data, and copies of our Chat application in separate sets of experiments.

The results show that migration latency increases linearly with the number of stored tuples or copies of Chat. We measure a throughput of 12.6 KB/second for tuples carrying 100 bytes of data, 16.2 KB/second for copies of Chat, and 1.557 KB/second for tuples carrying 100,000 bytes of data. In the best case (tuples carrying 100,000 bytes), migration utilizes 12% of the theoretically available bandwidth and is limited by how fast stored tuples can be moved from one PC to the other. Since moving a stored tuple requires reading the tuple from disk, sending it across the network, writing it to disk, and confirming its arrival, a better performing migration protocol than the one described in Appendix A should optimistically stream tuples and thus overlap the individual steps instead of moving one tuple per protocol round.

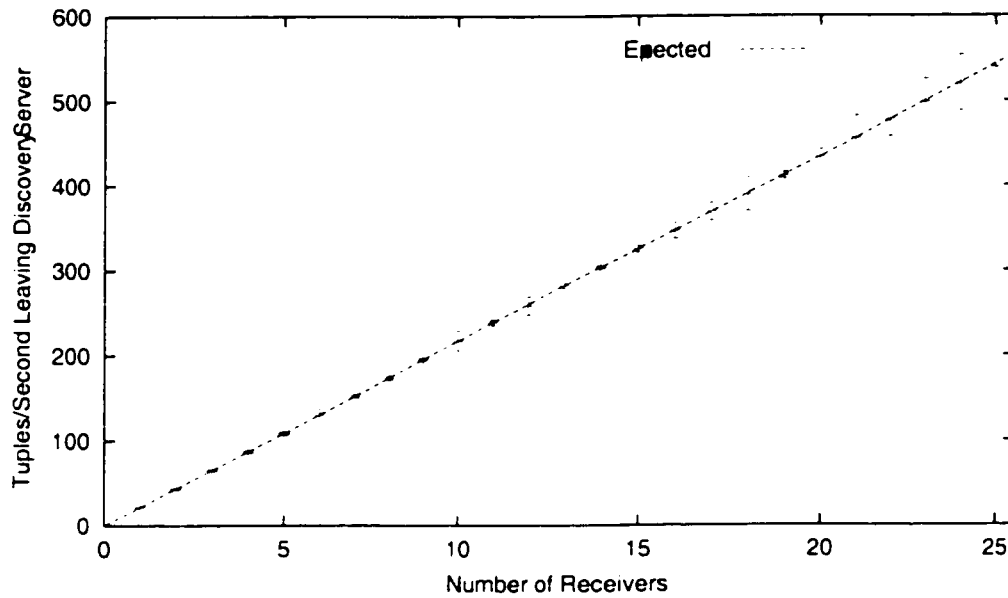


Figure 6.4: Discovery server throughput under an increasing number of receivers. Throughput is measured as the number of audio messages leaving the discovery server. The results shown are the average of 30 measurements, with error bars indicating the standard deviation. Each audio message carries 8 KB of audio data.

To quantify the scalability of late binding discovery, we stream audio messages between a varying number of Chat applications. We chose to measure streaming audio, because messages are large (see below) and must be delivered on time, thus exercising our implementation of the discovery service. Figures 6.4 and 6.5 show the discovery server throughput under an increasing number of receivers for a single sender and an increasing number of senders for a single receiver, respectively. Throughput is measured as audio messages leaving the discovery server, and the results shown are the average of 30 measurements. Each audio message carries 8 KB of uncompressed audio data at CD sampling rate, which corresponds to 10,118 bytes on the wire when forwarding from the sending node to the discovery server and 9,829 bytes when forwarding from the discovery server to the receiving node. The difference in on-the-wire sizes stems largely from the fact that messages forwarded to the discovery server contain the late binding query, while messages forwarded from the discovery server do not. The receivers and senders respectively run on four PCs; we use Emcee’s support for copying applications via drag and drop to spawn new ones.

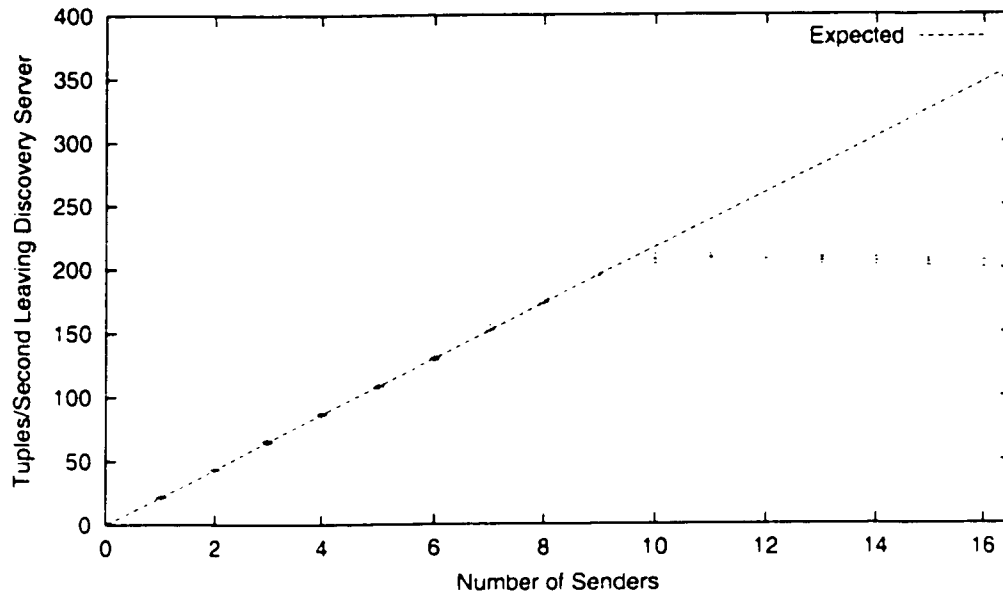


Figure 6.5: Discovery server throughput under an increasing number of senders. As in Figure 6.4, throughput is measured as the number of audio messages leaving the discovery server. The results shown are the average of 30 measurements, with error bars indicating the standard deviation. Each audio message carries 8 KB of audio data.

When increasing the number of receivers, discovery server throughput increases almost linearly with the number of receivers. However, when increasing the number of senders, discovery server throughput levels off at about 10 senders and slightly degrades thereafter. At 10 senders, the PC running the discovery server becomes CPU bound. While the cost of processing discovery queries remains low, the cost of processing UDP packets and serializing and deserializing audio messages comes to dominate that PC's performance.

Figure 6.6 illustrates system and application reactivity by showing the audio messages received by Chat as its runtime context changes. As for the discovery server throughput experiments, each audio message carries 8 KB of uncompressed audio data at CD sampling rate. Unlike the migration latency experiments, Chat is managed by Emcee and runs within its user's environment. At point 1, Chat is subscribed to an audio channel and starts receiving audio messages shortly thereafter. At point 2, Chat is moved to a different device and does not receive audio messages for 3.7 seconds while it migrates, reinitializes audio, and reregisters with discovery. After it has been migrated

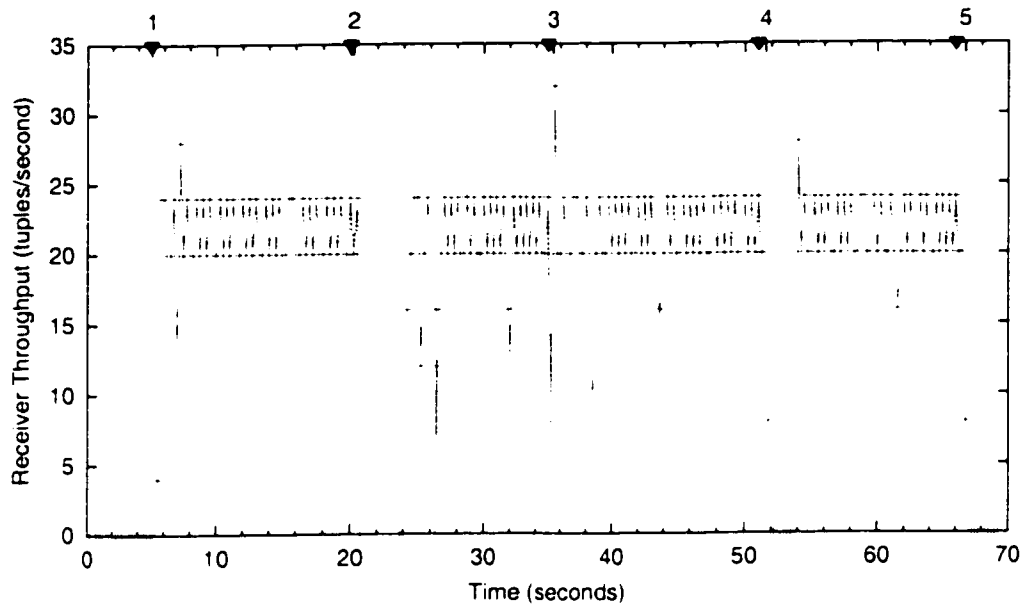


Figure 6.6: Audio messages received by Chat in a changing runtime environment. Chat is subscribed to an audio channel at point 1. It is then moved to a different node at point 2. The node hosting the discovery server is shut down gracefully at point 3 and forcibly crashed at point 4. The audio channel is unsubscribed at point 5.

and its receiving event handler has been reexported to discovery, it starts receiving audio messages again. The PC running the discovery server is gracefully shut down at point 3. Since *one.world* proactively calls a discovery server election, the stream of audio messages is not interrupted. By contrast, at point 4, the PC running the discovery server is forcibly crashed. The stream of audio messages is interrupted for 2.3 seconds until a new discovery server is elected and Chat's receiving event handler is forwarded to the new discovery server. This period is governed by detecting the crashed discovery server, which requires two missed server announcements or 2 seconds. Finally, at point 5, Chat is unsubscribed from the audio channel and stops receiving audio messages shortly thereafter.

Overall, our performance evaluation shows that service interruptions due to migration or forced discovery server elections last only a few seconds, which compares favorably with Microsoft's Outlook hanging for several minutes. Furthermore, while migration latency generally depends on the number and size of stored tuples, it takes only 7 seconds for an environment storing 8 MB of audio

data, which is fast enough when compared to a person moving through the physical world. Finally, our architecture performs well enough to support several independent streams of uncompressed audio data at CD sampling rate. However, our evaluation also suggests that discovery server scalability is limited. Adding a secondary discovery server, as already suggested in Chapter 5.3, could improve the scalability of our discovery service and would also eliminate service interruption due to forced server elections.

6.4 Utility

To determine utility, we used *one.world* as the basis for student projects in our course on building distributed and pervasive applications and supported the Labscape project in porting their digital laboratory assistant to our architecture. We now discuss these applications, presenting the course projects first and Labscape second.

6.4.1 Building Distributed and Pervasive Applications

Relatively early in our implementation effort (corresponding to releases 0.1 through 0.4 of our architecture) we conducted an experimental comparison of *one.world* with other distributed systems technologies by teaching a senior-level capstone design course. The nine students in the class split into two teams that each implemented a distributed application. Each team, in turn, split into two subteams, with one subteam using existing Java-based technologies and the other using *one.world*. Since both subteams implemented the same application, this experiment lets us compare our architecture with other approaches to building distributed systems. To document the design and implementation process and to compare the different application versions, each team produced a web-based diary, a design report, and a final report. Students also attended weekly meetings with the instructors and gave final presentations about their work. Finally, another researcher, Vibha Sazawal, conducted end-of-term interviews with the students, focusing her questions on software engineering issues. The results reported here are based on the teams' documentation and end-of-term interviews.

The first team developed a music sharing system, which relies on a dynamically configured hierarchy of directory nodes to organize searches. The Java subteam implemented the application

in plain Java, without using additional technologies. Results for the music sharing team are incomplete; students barely completed the implementations, although they did demonstrate working applications. The students' experiences suggest that our architecture's support for queries as part of structured I/O and for asynchronous messaging through REP clearly simplified the implementation on top of *one.world*. In contrast, the Java subteam had to implement querying and asynchronous messaging from scratch.

The second team developed a universal inbox, which integrates a home network of future smart appliances, such as an intelligent fridge, with email access from outside the network, thus making it possible, for example, to control one's appliances while traveling. The universal inbox lets users access human-readable email, routes control messages to and from appliances, and provides a common data repository for email and appliance configuration state. The goal was to build a reliable application that gracefully reacts to changes in the runtime environment, such as a computer crash. The Java version uses Jini [5] for service configuration and T Spaces [116] for storing repository data.

The students' experiences support our argument from Chapter 2 that extending programming models for single-node applications to distributed systems makes it difficult to build adaptable applications. Jini relies on Java RMI to access remote resources and is designed to simplify the conversion of existing code into Jini services. The Java subteam exploited this and originally implemented individual services, such as the message router or data repository, as stand-alone, single-node applications. Students subsequently "jinified" the applications and iteratively refined them as network services. While the conversion of an application into a bare-bones Jini service is simple, turning a minimal Jini service into a full-blown Jini service is an arduous process. This refinement process involved repeatedly testing the system to identify potential failure conditions and then adding code to account for such conditions. Students also had to work around the synchronous design of RMI. While Jini includes support for remote events, they are implemented as synchronous invocations through RMI and thus expose services to possibly indefinite delays, for example, because the service receiving an event is buggy and hangs. The completed implementation still reflects the difficulties of the refinement process and has relatively few services, with each of these services representing a single point of failure for all users.

In contrast, it appears that the *one.world* subteam did not encounter similar difficulties in build-

ing their version. Rather, local tuple storage and late binding seem to have simplified the *one.world* version. More specifically, their implementation avoids a centralized data repository and, by building on local tuple storage, separates each user's email management into an independently running service. The individual services communicate with each other through late binding discovery, thus masking their current location as well as transient failures. As a result, the implementation running on our architecture is more resilient to failures and more adaptable.

6.4.2 *Labscape*

We now return to the Labscape digital biology laboratory assistant, which we introduced in Chapter 2.3. Remember that the goal is to seamlessly capture, organize, and present biology processes in order to help biologists perform reproducible experiments. The Labscape application tries to achieve this goal by having an experimental guide follow a researcher from touchscreen to touchscreen as she moves through the laboratory. The constraints are that (1) the Labscape developers are programmers and not system builders, and (2) the resulting application has to be good enough to be used by real biologists every day. In other words, the Labscape application has to be responsive, stable, and robust. The application needs to react quickly to changes in its execution context, and it needs to be continuously available. Furthermore, when a failure occurs, its effects should be localized and it should be easy to recover.

The Labscape team actually created three different implementations of their digital laboratory assistant. The first version centralizes all processing and relies on remote windowing through VNC [93] to display the individual guides on a laboratory's touchscreens. In the Labscape team's experience, the first version is neither responsive nor robust. While a different remote windowing system, such as X Windows [82], might have alleviated the performance concerns, it would not have eliminated the central point of failure. Furthermore, the first version's reliance on remote windowing precludes more advanced features, such as a researcher reviewing her work while commuting homewards and being disconnected from the digital laboratory.

The second version of Labscape uses distributed processing; code and data follow the researchers through the laboratory by migrating from touchscreen to touchscreen. The Labscape team implemented the second version directly in Java, using TCP sockets for communications and their own,

application-specific migration layer to move the guides. In their experience, the second version is neither stable nor robust, thus prompting the Labscape team to port their application to *one.world*. This third version of Labscape has the same basic structure as the second version. However, it relies on our architecture's late binding discovery for communications and the migration service for moving guides between touchscreens. The resulting application is responsive, stable, and robust.

The structure of the second and third versions is illustrated in Figure 6.7. The individual application services work as following. The *device access service* collects experimental data from RFID and barcode scanners and location updates from IR sensors. It converts the data and the updates into the appropriate events and then forwards them to the *proximity service*, which tracks researchers' locations. For experimental data, the proximity service determines the researcher that performed the scanning operation and, in turn, forwards the data to the researcher's guide. For location updates, the proximity service updates its internal data structures and then advises the researcher's guide to move to the closest touchscreen. The *WebDAV service* is used to publish experimental data on the World Wide Web. Finally, the *state service* serves as the final repository for all experimental data, which it receives from the researchers' guides.

Porting to *one.world* resulted in three major benefits over the Java version. First, it reduced the development time from nine to four man months. In part, the reduced development time stems from the fact that the Labscape team did not have to redesign their application and could reuse existing code, as our architecture remains neutral on an application's structuring (as stipulated in Chapter 3). Second, porting simplified code maintenance and improved performance. In the Java version, every major modification of the guide requires corresponding changes in the migration layer. Yet, despite the application-specific migration layer, moving a guide in the Java version is five to ten times slower than in the *one.world* version. In [6], the Labscape team reports that migration latencies for the *one.world* version are between 2.5 seconds for moving a guide with no experimental data and 7.1 seconds for moving a guide with 64 samples, representing a large experiment. These results are consistent with our own measurements, as reported above, and are acceptable when compared to a researcher moving through the laboratory—while migration latencies around a minute are not.

Finally, porting considerably improved application uptime and resilience to failures. The Java version has a mean time between failures (MTBF) of 30 minutes, compared to an order of days for the *one.world* version. The short MTBF of the Java version stems from a lack of appropriate

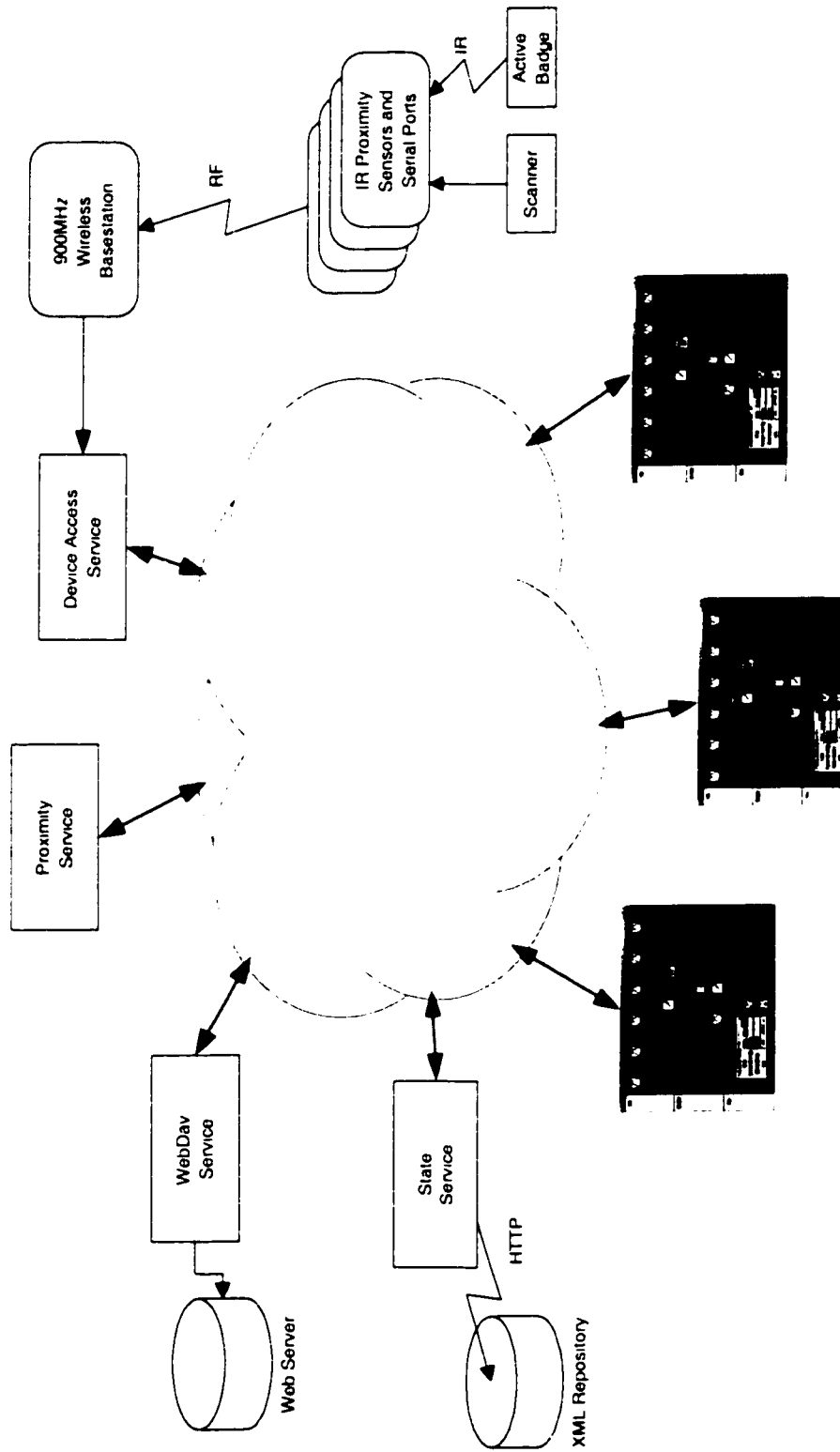


Figure 6.7: Illustration of Labscape's structure (second and third version). The experimental guides (shown at the bottom) use migration to follow researchers as they move through the laboratory and communication services. The second version is implemented in plain Java and relies on an application-specific migration layer as well as TCP sockets. In contrast, the third version is implemented on top of *one.world* and uses our architecture's migration service and late binding discovery. The individual application services are explained in the text.

system support as well as from buggy application code. Porting to *one.world* can eliminate the first cause but not the second cause. At the same time, system support can help with graceful failure recovery. In particular, after a failure of the Java version, the *entire* digital laboratory has to be restarted. In contrast, by building on late binding discovery instead of direct TCP connections, the *one.world* version allows for a piecemeal restarting of application services and thus is considerably more resilient in the face of buggy application code.

Once more, the Labscape application illustrates the power of combining *one.world*'s migration with late binding discovery. As programs move from one device to another, they easily communicate with each other by routing events through the discovery service. At the same time, Labscape's use of migration differs from the other programs discussed in this chapter. Unlike the replication service—which uses migration as an internal building block—and Emcee—which controls how other applications are migrated, the guides in Labscape migrate themselves. Given this versatility of migration, we believe it to be a general building block for system services, utilities, and applications alike.

To improve their application's resilience to failures, the Labscape team plans to further exploit *one.world*'s features. In particular, they intend to replicate the proximity service, which currently represents a single point of failure. The corresponding code changes are simple, as they only require changing event delivery from anycast to multicast and ignoring duplicate events in the guides. Furthermore, the Labscape team plans to add support for disconnected operation to the guides. Currently, the guides require that the state service be continuously available, so that they can directly forward updates. Support for disconnected operation can easily be added by logging pending updates in local tuple storage and forwarding them once the state service becomes available again.

Overall, the Labscape application demonstrates that *one.world* provides a solid platform for building and running real pervasive applications. However, the Labscape team did encounter three limitations of our architecture. First, in the Labscape team's experience, *one.world* events are harder to program than, for example, Java Swing events. In particular, they would like to write more concise event code and see better support for managing asynchronous interactions (in addition to our operation library). The students implementing the universal inbox for our course on building distributed and pervasive applications also voiced similar concerns. Second, *one.world* has its own data model based on tuples and its own network communications in the form of REP and discovery.

As a result, it is unnecessarily hard to interact with legacy and web systems. We revisit both issues in more detail in the next chapter.

Finally, due to security constraints enforced by our architecture, the Labscape team had difficulties in reusing existing, third-party Java libraries. More specifically, our architecture prevents applications from accessing Java's `java.lang.System` class and makes select methods, notably `arraycopy()` to copy the contents of arrays and `getProperty()` to access system properties, accessible through its own `one.world.util.SystemUtilities` class. Using a different class to access these methods does not represent a restriction for applications written from scratch: developers simply use a different class name in the source code. However, it does prevent existing Java libraries, which frequently employ these methods, from running on *one.world*. To address this issue, we developed a simple utility that, through binary rewriting, transforms existing libraries and replaces invocations to `System`'s methods with the corresponding *one.world* methods.

Chapter 7

DISCUSSION AND FUTURE WORK

As shown in the previous chapter, the user-space programs we and others built provide us with a solid basis for evaluating *one.world*'s design and implementation. The process of developing and using these programs also helped us gain a better understanding of the strengths and limitations of our architecture and its implementation. In this chapter, we focus on the resulting insights and identify lessons that are applicable beyond our work as well as opportunities for future research into system support for pervasive applications.

The user-space programs presented in Chapter 6 make extensive use of *one.world*'s services and illustrate the power of a design that follows the three requirements of change, ad hoc composition, and pervasive sharing:

Embrace contextual change. Event-based notification cleanly exposes change to applications. For example, the Labscape application uses events to expose location changes to a researcher's guide, enabling that guide to move to a close-by touchscreen. Furthermore, Chat relies on events to automatically adjust its configuration when the user owning the application changes.

Encourage ad hoc composition. Environment nesting and discovery make it easy to dynamically compose functionality. For example, Emcee relies on environment nesting to control a user's applications, and both our debugger and replication service use the request/monitor mechanism to interpose on an application's request stream. Furthermore, Emcee, Chat, the universal inbox, and the Labscape application all rely on discovery to connect different application instances or services. Moreover, discovery not only simplifies communications in the face of migrating applications—as is the case for Chat—but also increases applications' resilience to failures—as illustrated by Labscape.

Recognize sharing as a default. Tuples simplify the capturing and searching of information. For

example, the Labscape application directly encodes experimental data as tuples, and our students' music sharing system directly builds on our query language to locate appropriate songs. Furthermore, the separation of data—in the form of tuples—and functionality—in the form of components, provides considerable flexibility when compared to systems that combine data and functionality in objects. For instance, we can add music to a running Chat application, simply by importing the corresponding files into Chat's environment. Conversely, we can improve existing audio capabilities by instantiating the corresponding components in Chat's environment. Yet, while upgrading the application, we do not need to change stored audio tuples.

Additionally, migration and remote event passing provide powerful primitives that cover the spectrum between collocation and remote interaction. On one side, we rely on migration to make a user's applications available on a close-by device. On the other side, we rely on REP to let applications communicate with each other.

The central role played by environments in our architecture implies, in our opinion, a more general pattern, namely that *nesting is a powerful paradigm for controlling and composing applications*. To reiterate, nesting provides control, as illustrated by Emcee, and nesting can be used to extend applications, as illustrated by our replication service. Nesting thus makes it possible to easily factor important and possibly complex behaviors and provide them as common application building blocks. Furthermore, nesting is attractive because it preserves the relationships between nested environments. For instance, when audio tuples are stored in a child environment of Chat's environment, the environment with audio tuples remains a child, even if Chat's environment is nested in a user's environment and subsequently moved between devices.

While the user-space programs provide ample examples for the power of our architecture, they also helped in identifying several limitations. We discuss the issues raised by our data model in Chapter 7.1, followed by event processing in Chapter 7.2, leases in Chapter 7.3, and structured I/O's unified interface to storage and communications in Chapter 7.4. We then discuss user interfaces in Chapter 7.5 and the interaction between *one.world* and the outside world in Chapter 7.6. We conclude this chapter with an outlook on future research directions in Chapter 7.7.

7.1 Data Model

The biggest limitation of our architecture is that, to access a tuple, a component also needs to have access to the tuple's class. This does not typically pose a problem for applications, which have access to their own classes. However, it does pose a problem for services, such as discovery, that process many different types of data for many different applications. One solution, which we have not yet implemented, uses a generic tuple class, say `StaticTuple`, to provide access to the fields of different classes of tuples by using the accessor methods shown in Figure 4.1 on page 31. When passing a tuple across protection domains or when sending it across the network, the system tries to locate the tuple's class. If the class can be accessed, the tuple is instantiated in its native format. If the class cannot be accessed, the tuple is instantiated as a `StaticTuple`. In contrast to the `DynamicTuple` described in Chapter 4.2, whose fields are dynamically added and removed as well as dynamically typed, a `StaticTuple` preserves all typing information of a tuple's original class. In particular, it ensures that field values conform with the fields' declared types, and it does not support the dynamic addition/removal of fields. This solution works because services that process many different types of data already use the accessor methods instead of accessing a tuple's fields directly.

A `StaticTuple` can provide access to a tuple's fields even if the tuple's class cannot be accessed. At the same time, it cannot capture the semantic constraints expressed by the tuple's `validate()` method or the human-readable representation expressed by the `toString()` method. As a result, it represents a workable yet incomplete solution. The fundamental problem is that we have taken a single-node programming methodology, namely a programmatic data model, which expresses data schemas in the form of code, and applied it to a distributed system. This suggests that we need to abandon the programmatic data model altogether and instead use a declarative data model, which expresses schemas as data and not as code. With a declarative data model, applications still need to access a data item's schema in order to manipulate the data item. However, since the schemas themselves are data and not code, they are easier to inspect programmatically and not tied to a specific execution platform. As a result, we conclude that *declarative data models provide better interoperability than programmatic data models*.

We believe that defining an appropriate declarative data model is an important topic for future

research into pervasive computing. The challenge is to define a data model that meets conflicting requirements. On one side, to support the pervasive sharing of information, the data model must be general and supported by a wide range of platforms. One possible starting point is XML Schema [10, 105]. It already defines the data model for SOAP [14], which is the emerging standard for remote communications between web services and used, for example, by Microsoft's .NET platform [104]. On the other side, the data model must be easy to program and efficient to use. For an XML-based data model, this means avoiding the complexities of a general data access interface, such as DOM [64], and providing a more efficient encoding, perhaps by using a binary encoding [72] or by compressing the data [69]. Ideally, a declarative data model should be as easy to program as field access for tuples in our architecture. Probably, such a data model will specify a generic data container and a provision for automatically mapping data to application-specific objects, comparable to our proposed use of `StaticTuple`.

While tuples are limited by being based on a programmatic data model, the uniform and ubiquitous use of tuples in our architecture has proven to be very powerful. In particular, it allowed us to gracefully evolve the discovery service and integrate new functionality not found in other discovery systems. The initial design of our discovery service did not include support for reverse lookups (as described in Chapter 4.4). However, while implementing Chat, we needed some means for debugging remote communications through late binding discovery. We considered adding a dedicated interface for debugging discovery, but rejected that option as not general enough. We then converged on reverse lookups as a more flexible technique. Because of our architecture's uniform use of tuples, integrating reverse lookups with discovery was easy. Since events are tuples, they can be treated just like any other data, and reverse lookups on events can be directly expressed as regular queries. Furthermore, since queries are tuples, we simply added one more option to the *export* operation. We thus conclude that *the uniform use of structured data enables new functionality and helps to gracefully evolve a system.*

7.2 Event Processing

Asynchronous events provide a good fit for pervasive applications, as they make changes in an application's execution context explicit. However, event-based programs, unlike thread-based programs,

cannot store relevant state on the stack, thus raising the question of how to maintain the state associated with event exchanges. In our experience, two techniques have proven effective. First, we rely on state objects in event closures to establish relevant context. For example, Emcee already needs to maintain an internal table of user records, listing, among other things, a user's name and root environment. Emcee includes this user record as the closure for any request sent to an operation (which immediately replaces that closure as described in Chapter 5.2). The code processing responses can then determine the appropriate context based on the closure returned with the response. Second, when performing several related operations, we rely on a worklist that is maintained by the event handler receiving responses, e.g., the continuation event handler in Figure 4.7 on page 41. Upon receiving a response, the continuation removes the next item from the worklist and initiates the corresponding operation if the worklist has more items, or it invokes the appropriate logic if the worklist is empty. For instance, after activation, restoration, or migration, Chat uses such a worklist, with each item on the worklist describing a channel, to export an event handler to discovery for every subscribed channel.

Several event handlers in the programs described in Chapter 6 need to process many different types of events or perform different actions for the same type of event depending on the event's closure. Their implementation requires large if-then-else blocks that use `instanceof` tests to dispatch on the type of event or more general tests to dispatch on the value of the closure. The result is that these event handlers are not very modular and are relatively hard to understand, modify, or extend—an issue expressly noted by the Labscape team and the students implementing the universal inbox. This suggests the need for better programming language support to structure event handlers. Alternatives include dynamic dispatch as provided by MultiJava [24] or pattern matching as provided by Standard ML [76].

While we still believe that asynchronous events are an appropriate abstraction for pervasive computing, our experience with event-based programming also suggests that, contrary to [85], *asynchronous events are as hard to program as threads*. Just like threads, asynchronous events can result in complex interactions between components. For example, a better performing alternative to the migration protocol described in Chapter 5.4 and measured in Chapter 6.3 might optimistically stream tuples rather than waiting for an acknowledgement for each tuple. However, providing flow control for streamed events can easily replicate the full complexity of TCP's flow control [99]. Furthermore,

just as a system can run out of space for new threads, event queues can run out of space for new events. Finally, asynchronous events are not a panacea and some interactions must be synchronous. For example, timers to detect lost events must be scheduled synchronously because scheduling them asynchronously would use the same mechanism whose failure they are meant to detect.

7.3 Leases

As described in Chapter 4.2 and Chapter 4.4, resource access in our architecture is leased. Leases provide an upper bound on the time resources can be accessed, although leases can still be revoked by *one.world*'s kernel before their expiration, notably when an application is migrated. To make the use of leases practical, we introduced a lease maintainer library early on in our implementation effort. The lease maintainer automatically renews the lease it manages until it is explicitly canceled. While lease maintainers work most of the time, they can still fail, allowing a lease to expire prematurely. For example, when a device is overloaded, lease renewal events may not be delivered on time. Furthermore, when a device, such as a laptop or handheld computer, is hibernating, renewal events cannot be delivered at all. As a result, applications need to be prepared to reacquire local resources, such as their environment's tuple storage, even though the resources are guaranteed to be available. We thus conclude that *leases do not work well for controlling local resources*. Instead, we prefer a simple bind/release protocol, optionally with callbacks for the forced reclamation of resources, and use leases only for controlling remote resources.

7.4 A Unified Interface to Storage and Communications

As described in Chapter 4.2.2, we took a cue from Unix and carefully designed structured I/O to expose the same basic interface for storage and communications (though, in contrast to tuple spaces, structured I/O storage and networking are distinct services). However, *none* of the programs we and others built actually use structured I/O networking; they all rely on remote event passing and discovery for network communications. Only REP and discovery themselves employ structured I/O networking to implement their services. We believe that developers favor REP and discovery over structured I/O networking for remote communications because the former are higher-level and more flexible services. As a result, we conclude that we overdesigned structured I/O. We could

have omitted structured I/O networking and instead used a simpler, internal networking layer for implementing REP and discovery. In other words, *storage and communications are orthogonal to each other and best implemented by separate services with distinct interfaces.*

7.5 User Interface

All GUI-based programs running on top of *one.world* use Java's Swing toolkit [112] to implement their user interfaces. The integration between Swing's event model and *one.world*'s event model has worked surprisingly well. When an application needs to react to a Swing event, it generates the corresponding *one.world* event and sends it to the appropriate event handler. Long-lasting operations, such as fetching a user's applications, are broken up into many different *one.world* events, which are processed by our architecture's thread pools. Swing's event dispatching thread, which executes an application's user interface code, is only used for generating the first *one.world* event in a sequence of *one.world* events. As a result, applications in our architecture, unlike other applications using Swing, do not need to spawn separate threads for processing long-lasting operations. In the opposite direction, when an application needs to update the user interface in reaction to a *one.world* event, it simply schedules the update through Swing's `SwingUtilities.invokeLater()` facility.

An important limitation of Swing and other, comparable toolkits is that the user interface does not scale across different devices. For example, we successfully used Emcee and Chat on tablet computers but would be hard pressed to also run them on, say, handheld computers. However, an important property of pervasive computing environments is the variety of supported devices (as illustrated in Figure 1.1 on page 1.1). While most of these devices rely on screens—albeit considerably smaller ones than those used with PCs—for output and some pointing device for input, some devices, such as Sony's Aibo robotic dog, employ entirely different forms of input and output, including speech. Consequently, we believe that an important topic for future research into pervasive computing is how to implement scalable user interfaces. One promising approach, which is being explored by the user interface markup language [1] (UIML) and the Mozilla project's XML-based user interface language [19] (XUL), is to define a declarative specification of an application's interface, which is automatically rendered according to a device's input and output capabilities.

An unexpected lesson relating to user interfaces is that *GUI-based applications help with the*

testing, debugging, and profiling of a system. Once we started using Emcee and Chat, we quickly discovered several bugs in our architecture that we had not encountered before. The two applications also helped us with identifying several performance bottlenecks in our implementation. We believe that this advantage of GUI-based applications stems from the fact that GUIs encourage users to “play” with applications. As a result, the system is exercised in different and unexpected ways, especially when compared to highly structured regression tests and interaction with a command line shell. Furthermore, it is easier to run many GUI-based applications at the same time and, consequently, to push a system’s limits.

7.6 Interacting with the Outside World

To provide its functionality, *one.world* prevents applications from using abstractions not defined by our architecture. By default, applications cannot spawn their own threads, access files, or bind to network sockets. These restrictions are implemented through a Java security policy [39]. As a result, specific applications can be granted access to threads, files, and sockets by modifying a device’s security policy. However, because these abstractions are not supported by our architecture, applications are fully responsible for their management, including their proper release when an application is migrated or terminated.

Access to sockets is especially important for applications that need to interact with the outside world, such as Internet services. For example, we have used a modified security policy to let a web server run in our architecture. The web server’s implementation is split into a front end and a plug-gable back end. The front end manages TCP connections, translates incoming HTTP requests into *one.world* events, and translates the resulting responses back to HTTP responses. It also translates between MIME data and tuples by relying on the same conversion framework used for translating between files and stored tuples. The default back end provides access to tuples stored in nested environments.

In the opposite direction, it is not currently practical for outside applications to communicate with *one.world* applications through REP or discovery, especially if the outside applications are not written in Java. Because of our programmatic data model, an outside application would have to reimplement large parts of Java’s object serialization, which is unnecessarily complex. How-

ever, to provide ubiquitous information access, pervasive applications must easily interact with each other, independent of the underlying systems platform, as well as with Internet services. After all, the Internet is the most successful distributed system, used by millions of people every day. We believe that moving to a declarative, XML-based data model, as discussed above, and using standardized communication protocols will help in providing better interoperability between pervasive applications, even if they run on different system architectures, and with Internet services. To put it differently, *modern distributed systems need to be compatible with Internet protocols first and offer additional capabilities second.*

7.7 Outlook

In addition to the work on declarative data models and more modular event handling suggested above, we see two major thrusts for future research into system support for pervasive applications, with the first aimed at creating more advanced system services and the second aimed at changing the way we build pervasive applications.

While *one.world*'s system services meet basic application needs, there are three areas for providing additional system services. First, we need better support for reflecting an application's runtime environment to the application, including an ontology for describing device characteristics, network connectivity, and location. For example, *one.world* provides only limited information about a device's capabilities (such as speed and memory capacity) and none about the current level of connectivity. Yet, remaining energy for battery operated devices and cost for network connectivity are important factors when deciding, for example, whether to migrate or to communicate. The main challenge here is to develop an appropriate ontology and the corresponding software sensors. The location stack [49] provides an excellent example for how to accomplish this for device and user locations.

Second, we need better support for synchronizing and streaming data between devices. While we have experimented with replication (as described in Chapter 6.1.1), our current implementation is not sufficiently tuned and not fully integrated with other system services. Furthermore, *one.world* does not provide any support for dynamically transforming data, for example, to reduce a video stream's fidelity when sending it across a low bandwidth cellular link. The main challenge here

is not how to synchronize data or transform streaming video, but rather how to provide a unified framework for both discrete data (such as experimental data or personal contacts) and streaming data (such as audio or video).

Third, we need better support for changing and upgrading an application's code. *one.world*'s migration can already be used to easily install an application on a new device by simply copying it. However, our architecture does not provide the ability to upgrade applications while they are running. Since many pervasive applications can be expected to be long running, being able to upgrade such applications without disrupting them is an important capability. The main challenge here is to design a mechanism that is automatic, secure, and general enough to also upgrade the system platform itself.

While system support, such as that provided by *one.world*, can simplify the task of developing pervasive applications, developers still need to program all application behaviors by hand. We believe that a higher-level approach is needed to help developers be more effective.

To this end, we turn to the web for inspiration and observe that it uses two related technologies, declarative specifications and scripting, to great effect. For example, web servers typically rely on declarative configuration files that specify how to map a server's virtual name space to actual resources and how to process and filter content. Similarly, web browsers rely on style sheets to specify the appearance of accessed web pages. In both cases, more advanced behaviors are typically expressed through scripts that are embedded in web pages. As already mentioned, Mozilla's XUL pushes these two technologies even further and relies on them to express an application's entire user interface.

Based on our experiences with *one.world*, which suggest that application-specific policies can be directly implemented on top of our architecture's services, we believe that a similar approach can be applied to the development of pervasive applications. For example, this approach can be used to specify policies for migrating pervasive applications or data integrity constraints for replicated storage. The key insight is that a declarative specification can provide a concise description of a system's properties, which can then automatically be translated into appropriate actions. This approach thus represents a push towards specifying a system's goals instead of programming its behaviors. In effect, it treats a pervasive systems platform, such as *one.world*, as the assembly language for implementing complex behaviors. As such, it holds the potential to significantly simplify

the development of complex systems.

Chapter 8

RELATED WORK

one.world incorporates several technologies that have been successfully used by other systems. The main difference is that our architecture integrates these technologies into a simple and comprehensive framework, with the goal of enabling applications to adapt to an ever changing computing environment. Furthermore, where necessary, our architecture does introduce new services. Most importantly, our environment service is unique in that it combines persistent storage and the management of computations into a single, hierarchical structure. Other innovations include our remote event passing and discovery services, which expose an integrated API that covers the spectrum of network communications options, our migration service, which makes migration in the wide area practical, and our operation library, which effectively manages asynchronous interactions. In this chapter, we highlight relevant systems and discuss their differences when compared to *one.world*. Note that we have already reviewed systems that adapt transparently in Chapter 2.2.

The environment service was inspired by the ambient calculus [20]. Similar to environments, ambients are containers for data, functionality, and other ambients, resulting in a hierarchical structuring. Unlike environments, which are used to implement pervasive applications, ambients are an abstraction in a formal calculus and are used to reason about mobile computations. The MobileSpaces agent system [95] also relies on a hierarchical structuring, where agents can be embedded within other agents. Like environments, MobileSpaces agents are migrated together with all nested agents. Unlike environments, MobileSpaces agents provide only limited isolation (an outer agent can directly access the objects of an inner agent), cannot interpose on the request stream of inner agents (as provided in *one.world* through the request/monitor mechanism), and do not include persistent storage.

Asynchronous events have been used across a wide spectrum of systems, including networked sensors [50], embedded systems [23], user interfaces [89, 112], and large-scale servers [42, 86, 114]. Out of these systems, *one.world*'s support for asynchronous events closely mirrors that of DDS [42]

and SEDA [114]. As a result, it took the author of this dissertation a very short time to reimplement SEDA's thread pool controllers in *one.world*. Our architecture also provides three important improvements over these two systems. First, in DDS and SEDA, the event passing machinery is exposed to application developers, and events need to be explicitly enqueued in the appropriate event queues. In contrast, *one.world* automates event passing through the use of wrapped event handlers. Second, DDS and SEDA have no facilities for isolating the different stages, which map to environments in our architecture, from each other. In contrast, our architecture, by combining elements from KaffeOS [7] and the J-Kernel [111] while also avoiding those systems' limitations, does provide isolation between environments. Third, DDS and SEDA lack support for structuring event-based applications beyond breaking them into stages. While stages represent a significant advance when compared to prior event-based systems, operations in *one.world* provide additional structure for event-based applications and simplify the task of writing asynchronous code.

Odyssey [81] relies on asynchronous notifications to expose contextual change to applications. It is based on a client/server model, where applications' access to services is mediated by the Odyssey runtime. Under this model, applications specify allowable fidelity ranges for the services they use. The runtime, in turn, relies on type-specific components to map these fidelity ranges to actual resources, for example, to select an appropriate resolution for streaming video. When a service cannot be provided within the requested fidelity range, for example, because of insufficient network bandwidth, the Odyssey runtime notifies the application through an upcall, thus allowing the application to select a different fidelity range. Odyssey's use of asynchronous upcalls for exposing contextual change is comparable to our architecture's use of events. However, Odyssey has been designed as a minimal extension to a traditional operating system (NetBSD). As a result, it is far less flexible in specifying what resources to access (it only supports file names) and in notifying applications of contextual change (it only supports a single upcall with three simple parameters). Furthermore, it lacks more advanced services that help applications adapt, such as our architecture's migration and discovery services. At the same time, Odyssey's framework for mapping fidelity ranges to actual resources based on a resource's type is complimentary to our own work.

Starting with Linda [21], tuple spaces have been used to coordinate loosely coupled applications [26, 37, 79, 116]. Departing from the original tuple space model, several of these systems support more than one global tuple space and may even be extended through application-specific

code, for example, to automatically synchronize a local and a remote tuple space. Our architecture's use of tuples differs from these systems in that, as discussed in Chapter 4.2.2, structured I/O storage is a separate service from communications—whether through structured I/O networking or through remote event passing and discovery—and more closely resembles a database interface than Linda's *in*, *out*, and *rd* operations. At the same time, applications that require a traditional tuple space can easily implement such a service on top of remote event passing and structured I/O storage. Linda's *out* and *rd* operations map directly to structured I/O's *put* and *read* operations (though, every tuple written through a *put* must have a fresh GUID as its ID). Linda's *in* operation can be implemented as a transactional *read* and *delete*.

Like tuple spaces, the information bus helps with coordinating loosely coupled services [83]. Unlike tuple spaces, it is based on a publish/subscribe paradigm and does not retain sent messages in storage. While its design is nominally object-based, data exchanged through the bus is self-describing and separate from service objects, comparable to the separation of data—in the form of tuples—and functionality—in the form of components—in *one.world*. The information bus dynamically matches senders and receivers based on so-called subjects. Subjects are hierarchically structured strings, similar to DNS names, and matching supports equality testing as well as wild-cards. Messages are published under specific subjects and then delivered to all interested parties. In its ability to deliver messages to receivers based on a property of the message, the information bus resembles our architecture's reverse discovery lookups. However, the information bus provides only a very limited form of reverse lookup and does not support forward lookups at all. Interestingly, the information bus also includes an option for point-to-point communications (albeit through synchronous remote method invocations), just like our architecture supports both point-to-point communications and the dynamic matching between senders and receivers.

On the surface, Sun's Jini [5] appears to provide many of the same services as our architecture. However, Jini embodies a fundamentally different approach to building distributed applications: it extends single-node programming methodologies, is strongly object-oriented, and relies on remote method invocations. As a result, Jini requires an overall well-behaved computing environment, and its services are rather limited when compared to the corresponding services in *one.world*. In particular, Jini requires a statically configured discovery server. Moreover, service objects accessible through discovery double as their own descriptors and can only be accessed through early binding

and simple equality queries. Furthermore, Jini does not provide isolation between applications running on the same device, thus making it impossible to terminate ill-behaved programs without terminating all programs on the device. Likewise, Jini synchronously sends remote events through Java RMI, thus exposing the sender to arbitrary delays on the receiving side. Finally, Jini relies on distributed garbage collection [90] (DGC) for controlling objects' lifetimes. However, the illusion of a global pool of objects provided by DGC is misleading. Objects can still be prematurely reclaimed, for example, when devices are disconnected for a sufficiently long time and DGC's internal leases expire. DGC also makes it unnecessarily hard to provide migration on top of Jini. Since DGC controls objects' lifetimes, a migration service cannot move objects without either proxying every remotely accessible object or being fully integrated with DGC's implementation.

In addition to Jini, the intentional naming system [2] (INS), the secure discovery service [25] (SDS), the service location protocol [46] (SLP), and universal plug and play [75] (UPnP) all provide the ability to locate resources by their descriptions. Out of these systems, INS comes the closest to our architecture's discovery service. Like *one.world*, INS supports early and late binding as well as anycast and multicast. Furthermore, comparable to the use of discovery server elections in our architecture, INS' servers automatically form an overlay network to route late binding messages; though, individual servers still need to be manually configured. SDS and SLP both explore how to secure service discovery. Additionally, SDS includes a mechanism for aggregating service descriptions into a global hierarchy of discovery servers. Both efforts are complimentary to our own work. Finally, UPnP is largely targeted at automatically connecting PCs and stand-alone devices, such as printers and displays. As a result, it supports only simple matching queries (comparable to subject matching for the information bus). At the same time, UPnP does include support for event-based notifications when a device's state changes. The main difference between these services and our own is that *one.world* integrates discovery with point-to-point communications, resulting in a simple and elegant API that covers the spectrum of remote communications options. Furthermore, our discovery service is the only one to support reverse lookups.

An exhaustive review of previous work on migration is beyond the scope of this dissertation; instead, we refer the reader to Milošević et al.'s excellent *Mobility* [77]. For the purposes of this discussion, we focus on three representative, "best-of-breed" systems that cover the most important points in the design space. The three systems are transparent process migration as provided by

Table 8.1: Comparison of representative migration services with *one.world*'s migration service.

	Sprite	Emerald	Aglets	<i>one.world</i>
Target network	Local network	Local network	Internet	Local and wide area networks
Execution environment	Single instruction set	Multiple instruction sets	Virtual machine	Virtual machine
Unit of migration	Process	Object (including all "attached" objects)	Agent	Environment
Control over migration	Parent process	Any object with reference	Any object with reference, security policy on receiving side	Environment and its parents, both on sending and receiving side
Transparency	Yes, every machine appears just like the "home" machine	Yes, but location is visible	No, only agent with internal objects is migrated	No, only environment with its contents is migrated
Execution model	Process-based	Thread-based	Thread-based	Event-based
Migrated execution state	Stack	Stack	None	Event queue
Integration with storage	Distributed file system	None	None, but ability to save agents	Local tuple storage

Sprite [28], object mobility as provided by Emerald [56, 98], and mobile agents as provided by Aglets [63]. Table 8.1 provides a comparison of these systems with *one.world*'s migration service.

Probably the most important differentiating factor between Sprite and Emerald on one side and Aglets and *one.world* on the other side is whether migration is transparent. Both Sprite and Emerald target local networks and, based on the assumption that such networks are well-maintained and experience few failures, seek to provide transparent migration. They thus use forwarding addresses [35] and residual dependencies [91] to hide the effects of location changes (though, location itself is visible in Emerald), which results in considerable system complexity. Sprite's transparent integration with distributed file storage and Emerald's transparent support for multiple instruction sets represent additional sources of complexity. In contrast, migration in Aglets and *one.world* is not transparent. They migrate only an agent or environment, respectively, and can thus avoid most of the complexity of the other systems. However, Aglets' functionality also is rather limited, as it does not migrate execution state (though, the Telescript mobile agent system [115] does migrate execution state) and does not integrate storage beyond the ability to save agents. *one.world* differs from Aglets in that it migrates both execution state and persistent data. Furthermore, it differs from all three systems in that the environment hierarchy provides a well-defined and clean model for controlling which environments to move away from a device and which migrating environments to accept on a device.

In [97], Snoeren et al. introduce a different form of migration, which moves the end-point of a TCP connection to a different device (without tearing down the connection). TCP end-point migration is orthogonal and complimentary to the migration services discussed above. At the same time, in our experience with *one.world*, pervasive applications rarely use point-to-point communications. Instead, they typically communicate through late binding discovery and may additionally use multicast to address several components at the same time. As a result, we believe that TCP end-point migration is not as useful for pervasive applications as the environment migration provided by *one.world*.

Several other projects are exploring aspects of systems support for pervasive applications. Notably, InConcert, the architectural component of Microsoft's EasyLiving project [18], provides service composition in a dynamic environment by relying on location-independent names and asynchronous events. Furthermore, iROS, the operating system for Stanford University's iRoom project [52], features an asynchronous event distribution system, a shared tuple space that not only

stores but also transforms data, and an automatic user interface generation system. An important common theme to these efforts and our own is the need for networked communications that are asynchronous and dynamically match senders with receivers. iROS and *one.world* also share their reliance on tuples for representing all data, including events. At the same time, we fundamentally differ in our approaches. The EasyLiving and iRoom projects seek to better integrate the applications running in a single, intelligent room. As a result, they reuse existing applications wherever possible and provide only as much system support as strictly necessary. In contrast, *one.world* has been designed from the ground up to meet the requirements of pervasive applications. Consequently, our architecture is more complete and more powerful, but also requires that applications be written from scratch.

Several efforts, including Globe [109], Globus [34], and Legion [67], explore an object-oriented programming model and infrastructure for wide area computing. They share the important goal of providing a common execution environment that is secure and scales across a global computing infrastructure. However, these systems are targeted at collaborative and scientific applications running on conventional PCs and more powerful computers. As a result, these systems are too heavy-weight and not adaptable enough for pervasive computing environments. Furthermore, as argued in Chapter 2, we believe that their reliance on RPC for remote communications and on objects to encapsulate data and functionality is ill-advised.

Chapter 9

CONCLUSIONS

In this dissertation, we have explored how to build pervasive applications. Pervasive, or ubiquitous, computing presents an attractive vision for the future of distributed computing, where devices are ubiquitous and seamlessly coordinate to help people in accomplishing their tasks. However, existing approaches to building distributed applications are unsuitable for realizing this vision. The problem is that they try to hide distribution and rely on technologies, such as remote procedure call packages, that extend single-node programming methodologies to distributed systems. Applications built on top of these technologies tend to be structured like single-node applications and assume an execution environment where resources are constant and continuously available. As a result, users must explicitly reconfigure their devices and applications every time the execution environment changes, which is tedious at best and antithetical to the vision of pervasive computing at worst.

To better realize the vision of pervasive computing, we have introduced a more suitable approach to building pervasive applications. Under this approach, system support exposes distribution rather than hide it. That way, applications can see contextual change and then adapt to it instead of forcing users to constantly reconfigure their systems. More formally, this dissertation has explored the hypothesis that, by focusing on the unique requirements of pervasive computing, system support lets applications instead of users adapt to change, yet does not place an undue burden on developers. In particular, system support needs to address the following three requirements. First, systems need to *embrace contextual change*, so that applications can implement their own strategies for handling changes. Second, systems need to *encourage ad hoc composition*, so that applications can be dynamically connected and extended in an ever changing runtime environment. Third, systems need to *recognize sharing as the default*, so that applications can make information accessible anywhere and anytime.

We have presented *one.world*, a system architecture for pervasive computing, that embodies this approach to building pervasive applications. Our architecture builds on four foundation services

that directly address the three requirements. First, a virtual machine provides a uniform execution environment across all devices and supports the ad hoc composition between applications and devices. Second, tuples define a common type system for all applications and simplify the sharing of data. Third, events are used for all communications and make change explicit to applications. Finally, environments host applications, store persistent data, and—through nesting—facilitate the composition of applications and services. On top of these foundation services, our architecture provides a set of system services that address common application needs, including discovery to locate resources across the network and migration to move or copy applications between devices.

We have validated our architecture by supporting the Labscape team in porting their digital biology laboratory assistant to our architecture, by using *one.world* as the basis for student projects in a senior-level capstone design course, and by developing our own programs—including a replication service, a user and application manager, and a text and audio messaging system. Our experimental evaluation has demonstrated that *one.world* (1) is sufficiently complete to support additional services, utilities, and applications on top of it, (2) is not significantly harder to program than with conventional programming styles, (3) has acceptable performance, with applications reacting quickly to change, and, most importantly, (4) enables others to successfully build pervasive applications. However, our experimental evaluation has also shown that the scalability of our implementation, notably that of service discovery, is limited, making it suitable only for pervasive computing environments with several dozens of people and devices. Yet, despite these performance concerns, our evaluation has demonstrated that *one.world* lets developers effectively build applications that adapt to change, thus confirming the hypothesis behind this dissertation.

This dissertation has made the following research contributions. First, we have presented a system architecture for pervasive computing, called *one.world*, that has been designed from the ground up to address the unique requirements of pervasive computing—embracing change, encouraging composition, and facilitating sharing. Second, we have introduced the environment service, which provides a nestable container for persistent storage and computations alike and thus makes it possible to easily group applications and their data as well as to compose different applications. Third, we have presented an elegant and flexible remote communications API that provides service discovery and point-to-point communications through only three simple operations. Fourth, we have described a migration service that leverages our architecture's environment service to strike a better

balance between the complexity of transparent process migration and the limited utility of most mobile agent systems and that makes migration in the wide area practical. Finally, we have introduced the logic/operation pattern as a new software pattern for structuring asynchronous applications and the corresponding library to simplify application development.

Based on our own and others' experiences with *one.world*, this dissertation has also identified important lessons, both positive and negative, that are applicable beyond this work. Notably, we have demonstrated that nesting is a powerful paradigm for controlling and composing applications and that the uniform use of structured data enables new functionality and helps to gracefully evolve a system. However, we have also found that—unlike our architecture, which defines its own communication protocols—modern distributed systems need to be compatible with Internet protocols first and offer additional capabilities second. Furthermore, we have found that asynchronous events are as hard to program as threads, that leases do not work well for controlling local resources, and that storage and communications are orthogonal to each other and best implemented by separate services with distinct interfaces.

Finally, this dissertation has suggested directions for future research on pervasive computing. Most importantly, we have identified the significance of declarative specifications both for defining a shared data model and for implementing actual application functionality. The main research challenge in using declarative specifications for defining a shared data model is to meet conflicting requirements. On one side, such a data model must be general and supported by a wide range of computing platforms. On the other side, the data model must also be simple so that it is easy to program and efficient to implement. The main research challenge in using declarative specifications for implementing application functionality is to develop new, declarative programming languages and the corresponding compiler infrastructure to translate such high-level specifications into programs running on a system architecture such as *one.world*. While these research challenges are considerable, we also believe that any solution has the potential to significantly simplify the development of complex systems. More information on *one.world*, including a source release, is available at <http://one.cs.washington.edu>.

BIBLIOGRAPHY

- [1] Marc Abrams and Jim Helms. User interface markup language (UIML), version 3.0. Draft specification. Harmonia, Inc., Blacksburg, Virginia, February 2002. Available at <http://www.uiml.org/specs/uiml3/DraftSpec.htm>.
- [2] William Adje-Winoto, Elliot Schwartz, Hari Balakrishnan, and Jeremy Lilley. The design and implementation of an intentional naming system. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 186–201, Kiawah Island Resort, South Carolina, December 1999.
- [3] James P. Anderson. Computer security technology planning study. Technical Report ESD-TR-73-51, Vol. I, Electronic Systems Division, Air Force Systems Command, Bedford, Massachusetts, October 1972. Also AD-758 206, National Technical Information Service.
- [4] Thomas E. Anderson, Michael D. Dahlin, Jeanna M. Neefe, David A. Patterson, Drew S. Roselli, and Randolph Y. Wang. Serverless network file systems. *ACM Transactions on Computer Systems*, 14(1):41–79, February 1996.
- [5] Ken Arnold, Bryan O’Sullivan, Robert W. Scheifler, Jim Waldo, and Ann Wollrath. *The Jini Specification*. Addison-Wesley, 1999.
- [6] Larry Arnstein, Robert Grimm, Chia-Yang Hung, Jong Hee Kang, Anthony LaMarca, Stefan B. Sigurdsson, Jing Su, and Gaetano Borriello. Systems support for ubiquitous computing: A case study of two implementations of Labscape. In *Proceedings of the 2002 International Conference on Pervasive Computing*, Zurich, Switzerland, August 2002.
- [7] Godmar Back, Wilson C. Hsieh, and Jay Lepreau. Processes in KaffeOS: Isolation, resource management, and sharing in Java. In *Proceedings of the 4th USENIX Symposium on Oper-*

ating Systems Design and Implementation, pages 333–346. San Diego, California, October 2000.

- [8] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. Lightweight remote procedure call. *ACM Transactions on Computer Systems*, 8(1):37–55, February 1990.
- [9] Brian N. Bershad, Stefan Savage, Przemysław Pardyak, Emin Gün Sirer, Marc E. Fiuczynski, David Becker, Craig Chambers, and Susan Eggers. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 267–284, Copper Mountain, Colorado, December 1995.
- [10] Paul V. Biron and Ashok Malhotra. XML schema part 2: Datatypes. W3C recommendation, World Wide Web Consortium, Cambridge, Massachusetts, May 2001.
- [11] A. D. Birrell, R. Levin, R. M. Needham, and M. D. Schroeder. Grapevine: An exercise in distributed computing. *Communications of the ACM*, 25(4):260–274, April 1982.
- [12] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.
- [13] Bluetooth SIG. Specification of the Bluetooth system. Specification Version 1.1, Bluetooth SIG, February 2002.
- [14] Don Box, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Frystyk Nielsen, Satish Thatte, and Dave Winer. Simple object access protocol (SOAP) 1.1. W3C note, World Wide Web Consortium, Cambridge, Massachusetts, May 2000.
- [15] Tim Bray, Jean Paoli, and C. M. Sperberg-McQueen. Extensible markup language (XML) 1.0. W3C recommendation, World Wide Web Consortium, Cambridge, Massachusetts, February 1998.

- [16] Eric A. Brewer, Randy H. Katz, Elan Amir, Hari Balakrishnan, Yatin Chawathe, Armando Fox, Steven D. Gribble, Todd Hodes, Giao Nguyen, Venkata N. Padmanabhan, Mark Stemm, Srinivasan Seshan, and Tom Henderson. A network architecture for heterogeneous mobile computing. Technical report, University of California at Berkeley, 1998. Available at http://daedalus.cs.berkeley.edu/new_papers/Barwan.pdf.
- [17] Per Brinch Hansen. The nucleus of a multiprogramming system. *Communications of the ACM*, 13(4):238–241, 250, April 1970.
- [18] Barry Brumitt, Brian Meyers, John Krumm, Amanda Kern, and Steven Shafer. EasyLiving: Technologies for intelligent environments. In *Proceedings of the 2nd International Symposium on Handheld and Ubiquitous Computing*, volume 1927 of *Lecture Notes in Computer Science*, pages 12–29, Bristol, England, September 2000. Springer-Verlag.
- [19] Vaughn Bullard, Kevin T. Smith, and Michael C. Daconta. *Essential XUL Programming*. John Wiley & Sons, July 2001.
- [20] Luca Cardelli. Abstractions for mobile computations. In Vitek and Jensen [110], pages 51–94.
- [21] Nicholas Carriero and David Gelernter. The S/Net’s Linda kernel. *ACM Transactions on Computer Systems*, 4(2):110–129, May 1986.
- [22] Anawat Chankhunthod, Peter B. Danzig, Chuck Needaels, Michael F. Schwartz, and Kurt J. Worrell. A hierarchical Internet object cache. In *Proceedings of the 1996 USENIX Annual Technical Conference*, pages 153–163, San Diego, California, January 1996.
- [23] Pai Chou, Ross Ortega, Ken Hines, Kurt Partridge, and Gaetano Borriello. ipChinook: An integrated IP-based design framework for distributed embedded systems. In *Proceedings of the 36th ACM/IEEE Design Automation Conference*, pages 44–49, New Orleans, Louisiana, June 1999.

- [24] Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. MultiJava: Modular open classes and symmetric multiple dispatch for Java. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications '00*, pages 130–145, Minneapolis, Minnesota, October 2000.
- [25] Steven E. Czerwinski, Ben Y. Zhao, Todd D. Hodes, Anthony D. Joseph, and Randy H. Katz. An architecture for a secure service discovery service. In *Proceedings of the 5th ACM/IEEE International Conference on Mobile Computing and Networking*, pages 24–35, Seattle, Washington, August 1999.
- [26] Nigel Davies, Adrian Friday, Stephen P. Wade, and Gordon S. Blair. L²imbo: A distributed systems platform for mobile computing. *Mobile Networks and Applications*, 3(2):143–156, August 1998.
- [27] Michael L. Dertouzos. The future of computing. *Scientific American*, 281(2):52–55, August 1999.
- [28] Fred Douglass and John Ousterhout. Transparent process migration: Design alternatives and the Sprite implementation. *Software—Practice and Experience*, 21(8):757–785, August 1991.
- [29] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole Jr. Exokernel: an operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 251–266, Copper Mountain Resort, Colorado, December 1995.
- [30] Jeffrey L. Eppinger, Lily B. Mummert, and Alfred Z. Spector, editors. *Camelot and Avalon: A Distributed Transaction Facility*. Morgan Kaufmann, 1991.
- [31] Mike Esler, Jeffrey Hightower, Tom Anderson, and Gaetano Borriello. Next century challenges: Data-centric networking for invisible computing. In *Proceedings of the 5th*

- ACM/IEEE International Conference on Mobile Computing and Networking*, pages 256–262. Seattle, Washington, August 1999.
- [32] Pat Ferguson, Gloria Leman, Prasad Perini, Susan Renner, and Girish Seshagiri. Software process improvement works! Technical Report CMU/SEI-99-TR-027, Carnegie Mellon University, Software Engineering Institute, November 1999.
 - [33] Bryan Ford, Mike Hibler, Jay Lepreau, Patrick Tullmann, Godmar Back, and Stephen Clawson. Microkernels meet recursive virtual machines. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, pages 137–151. Seattle, Washington, October 1996.
 - [34] Ian Foster and Carl Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, 1997.
 - [35] Robert J. Fowler. *Decentralized Object Finding Using Forwarding Addresses*. PhD thesis, University of Washington, December 1985. Also available as Technical Report UW-CSE-85-12-01.
 - [36] Armando Fox, Steven D. Gribble, Yatin Chawathe, Eric A. Brewer, and Paul Gauthier. Cluster-based scalable network services. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 78–91. Saint-Malo, France, October 1997.
 - [37] Eric Freeman, Susanne Hupfer, and Ken Arnold. *JavaSpaces Principles, Patterns, and Practice*. Addison-Wesley, 1999.
 - [38] Matthew Gast. *802.11 Wireless Networks: The Definitive Guide*. O'Reilly, April 2002.
 - [39] Li Gong. *Inside Java Platform Security—Architecture, API Design, and Implementation*. Addison-Wesley, June 1999.

- [40] Cary G. Gray and David R. Chenton. Leases: An efficient fault-tolerant mechanism for file cache consistency. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 202–210, Litchfield Park, Arizona, December 1989.
- [41] Jim Gray, Pat Helland, Patrick O’Neil, and Dennis Shasha. The dangers of replication and a solution. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 173–182, Montreal, Canada, June 1996.
- [42] Steven D. Gribble, Eric A. Brewer, Joseph M. Hellerstein, and David Culler. Scalable, distributed data structures for Internet service construction. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation*, pages 319–332, San Diego, California, October 2000.
- [43] Robert Grimm and Brian N. Bershad. Separating access control policy, enforcement and functionality in extensible systems. *ACM Transactions on Computer Systems*, 19(1), February 2001.
- [44] Robert Grimm, Janet Davis, Eric Lemar, Adam MacBeth, Steven Swanson, Steven Gribble, Tom Anderson, Brian Bershad, Gaetano Borriello, and David Wetherall. Programming for pervasive computing environments. Technical Report UW-CSE-01-06-01, University of Washington, June 2001.
- [45] Robert Grimm, Michael M. Swift, and Henry M. Levy. Revisiting structured storage: A transactional record store. Technical Report UW-CSE-00-04-01, University of Washington, April 2000.
- [46] Erik Guttman, Charles Perkins, John Veizades, and Michael Day. Service location protocol, version 2. RFC 2608, Internet Engineering Task Force, June 1999.
- [47] Mark Guzdial and Kim Rose, editors. *Squeak: Open Personal Computing and Multimedia*. Prentice Hall, 2002.

- [48] John S. Heidemann and Gerald J. Popek. File-system development with stackable layers. *ACM Transactions on Computer Systems*, 12(1):58–89, February 1994.
- [49] Jeffrey Hightower, Barry Brumitt, and Gaetano Borriello. The location stack: A layered model for location in ubiquitous computing. In *Proceedings of the 4th IEEE Workshop on Mobile Computing Systems and Applications*, pages 22–28, Callicoon, New York, June 2002.
- [50] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister. System architecture directions for networked sensors. In *Proceedings of the 9th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 93–104, Cambridge, Massachusetts, November 2000.
- [51] John Ioannidis and Gerald Q. Maguire, Jr. The design and implementation of a mobile inter-networking architecture. In *Proceedings of 1993 Winter USENIX Conference*, pages 491–502, San Diego, California, January 1993.
- [52] Brad Johanson, Armando Fox, and Terry Winograd. The interactive workspaces project: Experiences with ubiquitous computing rooms. *IEEE Pervasive Computing Magazine*, 1(2):67–74, April–June 2002.
- [53] Kirk L. Johnson, John F. Carr, Mark S. Day, and M. Frans Kaashoek. The measured performance of content distribution networks. In *Proceedings of the 5th International Web Caching and Content Delivery Workshop*, Lisbon, Portugal, May 2000. <http://www.terena.nl/conf/wcw/Proceedings/S4/S4-1.pdf>.
- [54] Michael B. Jones. Interposition agents: Transparently interposing user code at the system interface. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 80–93, Asheville, North Carolina, December 1993.
- [55] Anthony D. Joseph, Alan F. deLespinasse, Joshua A. Tauber, David K. Gifford, and M. Frans Kaashoek. Rover: A toolkit for mobile information access. In *Proceedings of the 15th*

- ACM Symposium on Operating Systems Principles*, pages 156–171, Copper Mountain Resort, Colorado, December 1995.
- [56] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.
 - [57] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Héctor Briceño, Russell Hunt, David Mazières, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie. Application performance and flexibility on exokernel systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 52–65, Saint-Malo, France, October 1997.
 - [58] James J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. *ACM Transactions on Computer Systems*, 10(1):3–25, February 1992.
 - [59] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Zhao. OceanStore: An architecture for global-scale persistent storage. In *Proceedings of the 9th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 190–201, Cambridge, Massachusetts, November 2000.
 - [60] Puneet Kumar and M. Satyanarayanan. Flexible and safe resolution of file conflicts. In *Proceedings of the 1995 USENIX Annual Technical Conference*, pages 95–106, New Orleans, Louisiana, January 1995.
 - [61] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.
 - [62] Leslie Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):18–25, December 2001.
 - [63] Danny B. Lange and Mitsuru Oshima. *Programming and Deploying Java Mobile Agents with Aglets*. Addison Wesley, 1998.

- [64] Arnaud Le Hors, Philippe Le Hégarret, Lauren Wood, Gavin Nicol, Jonathan Robie, Mike Champion, and Steve Byrne. Document object model (DOM) level 2 core specification. W3C recommendation, World Wide Web Consortium, Cambridge, Massachusetts, November 2000.
- [65] Paul J. Leach and Rich Salz. UUIDs and GUIDs. Internet Draft draft-leach-uuids-guids-01.txt, Internet Engineering Task Force, February 1998.
- [66] Eliezer Levy and Abraham Silberschatz. Distributed file systems: Concepts and examples. *ACM Computing Surveys*, 22(4):321–374, December 1990.
- [67] Mike Lewis and Andrew Grimshaw. The core Legion object model. In *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*, pages 551–561, Syracuse, New York, August 1996.
- [68] Sheng Liang and Gilad Bracha. Dynamic class loading in the Java virtual machine. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications '98*, pages 36–44, Vancouver, Canada, October 1998.
- [69] Hartmut Liefke and Dan Suciu. XMill: An efficient compressor for XML data. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 153–164, Dallas, Texas, May 2000.
- [70] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, second edition, 1999.
- [71] Barbara Liskov, Dorothy Curtis, Paul Johnson, and Robert Scheifler. Implementation of Argus. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, pages 111–122, Austin, Texas, November 1987.
- [72] Bruce Martin and Bashir Jano. WAP binary XML content format. W3C note, World Wide Web Consortium, Cambridge, Massachusetts, June 1999.

- [73] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.4BSD Operating System*. Addison-Wesley, 1996.
- [74] Microsoft Corporation. *Microsoft Office 2000 Resource Kit*. Microsoft Press, July 1999.
- [75] Microsoft Corporation. Understanding universal plug and play. White paper, Microsoft Corporation, Redmond, Washington, June 2000. Available at http://www.upnp.org/download/UPNP_UnderstandingUPNP.doc.
- [76] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, May 1997.
- [77] Dejan Milojević, Frederick Douglass, and Richard Wheeler, editors. *Mobility—Processes, Computers, and Agents*. ACM Press, Addison-Wesley, February 1999.
- [78] Lily B. Mummert, Maria R. Ebling, and M. Satyanarayanan. Exploiting weak connectivity for mobile file access. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 143–155, Copper Mountain Resort, Colorado, December 1995.
- [79] Amy L. Murphy, Gian Pietro Picco, and Grigora-Catalin Roman. Lime: A middleware for physical and logical mobility. In *Proceedings of the 21st IEEE International Conference on Distributed Computing Systems*, pages 524–533, Phoenix, Arizona, April 2001.
- [80] Athicha Muthitacharoen, Benjie Chen, and David Mazières. A low-bandwidth network file system. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 174–187, Banff, Canada, October 2001.
- [81] Brian D. Noble, M. Satyanarayanan, Dushyanth Narayanan, James Eric Tilton, Jason Flinn, and Kevin R. Walker. Agile application-aware adaptation for mobility. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 276–287, Saint-Malo, France, October 1997.
- [82] Adrian Nye, editor. *X Protocol Reference Manual*. O'Reilly, 4th edition, January 1995.

- [83] Brian Oki, Manfred Pfluegl, Alex Siegel, and Dale Skeen. The Information Bus — an architecture for extensible distributed systems. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 58–68, Ashville, North Carolina, December 1993.
- [84] Michael A. Olson, Keith Bostic, and Margo Seltzer. Berkeley DB. In *Proceedings of the FREENIX Track, 1999 USENIX Annual Technical Conference*, pages 183–192, Monterey, California, June 1999.
- [85] John Ousterhout. Why threads are a bad idea (for most purposes). <http://home.pacbell.net/ouster/threads.ppt>, January 1996. Invited Talk presented at the *1996 USENIX Annual Technical Conference*, San Diego, California.
- [86] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. Flash: An efficient and portable Web server. In *Proceedings of the 1999 USENIX Annual Technical Conference*, pages 199–212, Monterey, California, June 1999.
- [87] Przemysław Pardyak and Brian N. Bershad. Dynamic binding for an extensible system. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, pages 201–212, Seattle, Washington, October 1996.
- [88] Karin Petersen, Mike J. Spreitzer, Douglas B. Terry, Marvin M. Theimer, and Alan J. Demers. Flexible update propagation for weakly consistent replication. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 288–301, Saint-Malo, France, October 1997.
- [89] Charles Petzold. *Programming Windows*. Microsoft Press, 5th edition, November 1998.
- [90] David Plainfossé and Marc Shapiro. A survey of distributed garbage collection techniques. In *International Workshop on Memory Management*, volume 986 of *Lecture Notes in Computer Science*, pages 211–249, Kinross, Scotland, September 1995. Springer-Verlag.

- [91] Michael L. Powell and Barton P. Miller. Process migration in DEMOS/MP. In *Proceedings of the 9th ACM Symposium on Operating Systems Principles*, pages 110–119. Bretton Woods, New Hampshire, October 1983.
- [92] Lutz Prechelt. An empirical comparison of seven programming languages. *IEEE Computer*, 33(10):23–29, October 2000.
- [93] Tristan Richardson, Quentin Stafford-Fraser, Kenneth R. Wood, and Andy Hopper. Virtual network computing. *IEEE Internet Computing*, 2(1):33–38, January/February 1998.
- [94] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, November 1984.
- [95] Ichiro Satoh. MobileSpaces: A framework for building adaptive distributed applications using a hierarchical mobile agent system. In *Proceedings of the 20th IEEE International Conference on Distributed Computing Systems*, pages 161–168, Taipei, Taiwan, April 2000.
- [96] Marc Shapiro. Structure and encapsulation in distributed systems: The proxy principle. In *Proceedings of the 6th IEEE International Conference on Distributed Computing Systems*, pages 198–204. Boston, Massachusetts, May 1986.
- [97] Alex C. Snoeren, David G. Andersen, and Hari Balakrishnan. Fine-grained failover using connection migration. In *Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems*, San Francisco, California, March 2001.
- [98] Bjarne Steensgaard and Eric Jul. Object and native code thread mobility among heterogeneous computers. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 68–77, Copper Mountain Resort, Colorado, December 1995.
- [99] W. Richard Stevens. *TCP/IP Illustrated*, volume 1. Addison-Wesley, 1994.
- [100] Sun Microsystems. Java remote method invocation specification. Revision 1.8. Sun Microsystems, Palo Alto, California, 2002.

- [101] Ariel Tamches and Barton P. Miller. Fine-grained dynamic instrumentation of commodity operating system kernels. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation*, pages 117–130, New Orleans, Louisiana, February 1999.
- [102] Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, Mike J. Spreitzer, and Carl H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 172–183, Copper Mountain Resort, Colorado, December 1995.
- [103] Renu Tewari, Michael Dahlin, Harrick M. Vin, and Jonathan S. Kay. Design considerations for distributed caching on the Internet. In *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems*, pages 273–284, Austin, Texas, June 1999.
- [104] Thuan Thai and Hoang Lam. *.NET Framework Essentials*. O'Reilly, 2nd edition, February 2002.
- [105] Henry S. Thompson, David Beech, Murray Maloney, and Noah Mendelsohn. XML schema part 1: Structures. W3C recommendation, World Wide Web Consortium, Cambridge, Massachusetts, May 2001.
- [106] Tommy Thorn. Programming languages for mobile code. *ACM Computing Surveys*, 29(3):213–239, September 1997.
- [107] Patrick Tullmann and Jay Lepreau. Nested Java processes: OS structure for mobile code. In *Proceedings of the 8th ACM SIGOPS European Workshop*, pages 111–117, Sintra, Portugal, September 1998.
- [108] Mitch Tulloch. *Windows 2000 Administration in a Nutshell*. O'Reilly, February 2001.
- [109] Maarten van Steen, Philip Homburg, and Andrew S. Tanenbaum. Globe: A wide-area distributed system. *IEEE Concurrency*, 7(1):70–78, 1999.

- [110] Jan Vitek and Christian D. Jensen, editors. *Secure Internet Programming: Security Issues for Distributed and Mobile Objects*, volume 1603 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.
- [111] Thorsten von Eicken, Chi-Chao Chang, Grzegorz Czajkowski, Chris Hawblitzel, Deyu Hu, and Dan Spoonhower. J-Kernel: a capability-based operating system for Java. In Vitek and Jensen [110], pages 369–393.
- [112] Kathy Walrath and Mary Campione. *The JFC Swing Tutorial: A Guide to Constructing GUIs*. Addison-Wesley, July 1999.
- [113] Mark Weiser. The computer for the twenty-first century. *Scientific American*, 265(3):94–104, September 1991.
- [114] Matt Welsh, David Culler, and Eric Brewer. SEDA: An architecture for well-conditioned, scalable Internet services. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 230–243, Banff, Canada, October 2001.
- [115] James E. White. Mobile agents. In Milojević et al. [77], pages 460–493.
- [116] Peter Wyckoff, Stephen W. McLaughry, Tobin J. Lehman, and Daniel A. Ford. T Spaces. *IBM Systems Journal*, 37(3):454–474, 1998.

Appendix A

THE MIGRATION PROTOCOL

The implementation of *one.world*'s migration service is centered around the protocol that communicates the migrating environment and its contents, including the execution state and all stored tuples, from the sending to the receiving device. Chapter 5.4 provides an overview over our architecture's migration protocol; this appendix presents the protocol in detail. Remember that the protocol moves or copies an entire environment tree in one, atomic operation and is implemented using our architecture's remote events. It is organized into several rounds, where each event issued by the sender is confirmed by the receiver, thus providing a very simple form of flow control. Both sender and receiver use operations to manage the event interchanges, employing a `ChainingClosure` as described in Chapter 5.2. The sender's operation connects each request to its response, while the receiver's operation connects each response to the next request. If an event is lost or an error occurs, the protocol and consequently the migration are aborted. The migration protocol is illustrated in Figure A.1; we use the step numbers in the text to refer back to this figure.

After receiving a migration request from an application, the sender prepares for the move or copy operation by quiescing all environments to be migrated and captures a checkpoint of their application state (step 1). Note that, on copy operations, the copied environments are assigned freshly generated GUIDs to avoid duplicate identifiers, although the human-readable names remain the same. After capturing the checkpoint, the sender issues the first protocol message, which specifies the name of the migrating environment and the identity of the new parent (step 2). It then sends the metadata for the migrating environment tree (step 5). Next, it reads and sends all stored tuples (steps 8 and 9). Note that, to avoid the performance overhead of serialization, tuples are not deserialized and serialized when being passed between storage and the network. Rather, they are directly forwarded as bytestrings that are contained in `BinaryData` tuples. Finally, the sender issues the checkpoint (step 12). Once the checkpoint has been confirmed, the migration protocol has successfully completed, and the sender can clean up its internal state (step 15). In particular, it destroys the

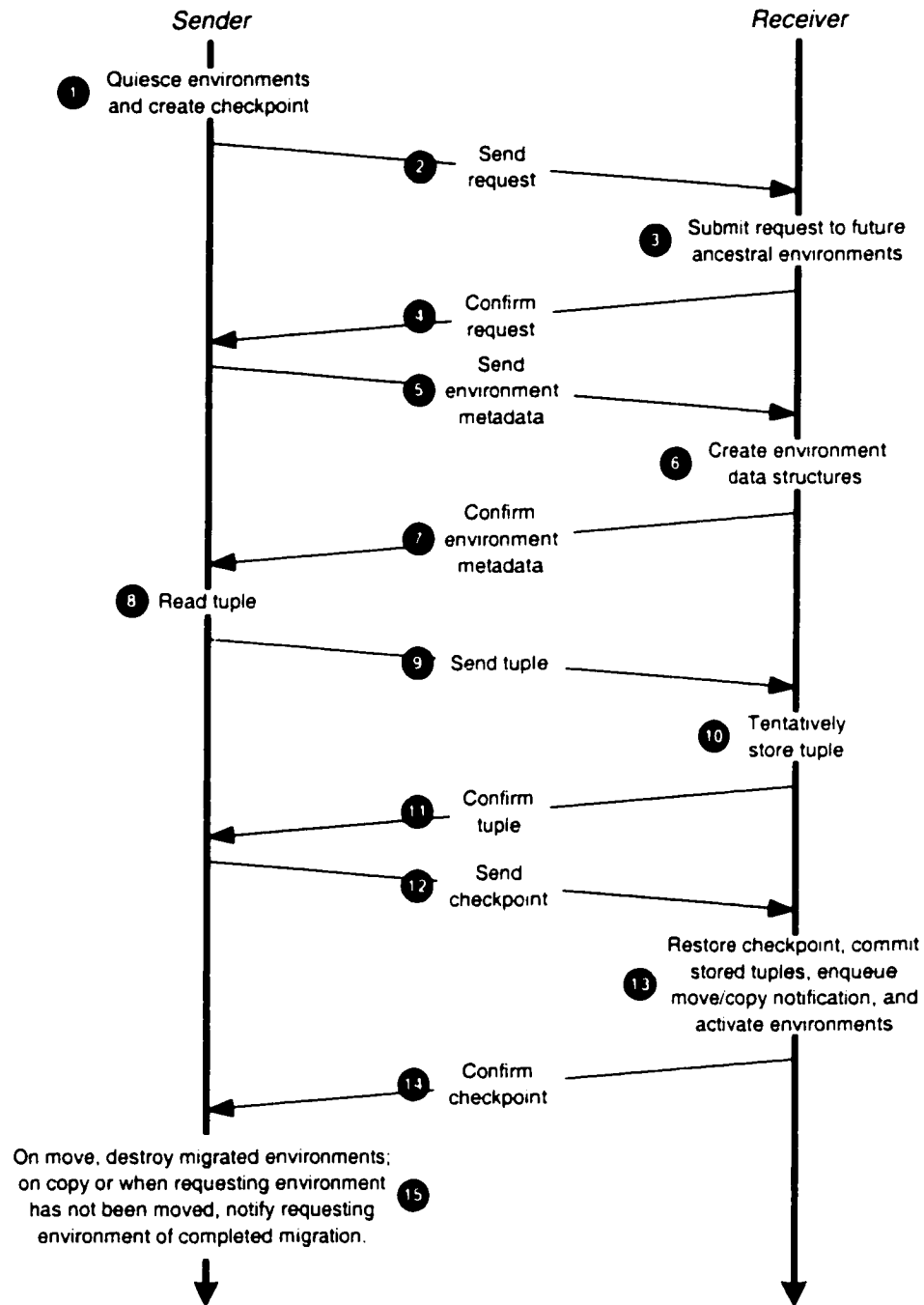


Figure A.1: Illustration of *one.world*'s migration protocol. The vertical arrows represent the flow of time. The slanted arrows represent messages. Note that steps 8, 9, 10, and 11 are performed zero or more times, depending on the number of tuples stored in the migrating environment tree.

migrated environment tree on a move operation, which invalidates references from the outside into the migrated environment tree. Furthermore, on a copy operation or if the requesting environment is outside the moved tree, the sender notifies the requesting environment of the successfully completed migration.

When receiving a migration request, the receiver uses the request/monitor mechanism to submit the request to all future ancestral environments, which can either redirect the migrating environment tree or reject it (step 3). Note that, in the current implementation, requests are accepted by default. In contrast, a production system should reject migration requests by default and, for obvious security reasons, only accept properly authenticated environments. Once accepted, the receiver confirms the request to the sender (step 4). After receiving the metadata for the migrating environment tree, the receiver creates the appropriate internal data structures (step 6) and issues a confirmation (step 7). Next, after receiving a tuple, the receiver tentatively stores that tuple (step 10) and confirms that it has received the tuple (step 11). Finally, after receiving the checkpoint, the receiver restores the checkpoint (step 13). Once the checkpoint has been restored, the migration has succeeded. The receiver commits all stored tuples, enqueues notifications of the completed migration, activates the migrated environments, and finally sends confirmation to the sender (step 14), which completes the migration protocol for the receiver.

VITA

Robert Grimm was born (in 1968) and raised in Germany. After working with mentally ill people and motion picture production staff for several years, he was ready to move to the U.S.A. in 1991. He earned a Bachelor of Science in Computer Science and Engineering and a Master of Engineering in Electrical Engineering and Computer Science from the Massachusetts Institute of Technology in 1996. He earned a Doctor of Philosophy in Computer Science from the University of Washington in 2002.